Reference

GET-HELP ABOUT

Get-help about

12/9/2018

about_aliases

SHORT DESCRIPTION

Describes how to use alternate names for cmdlets and commands in Windows PowerShell.

LONG DESCRIPTION

An alias is an alternate name or nickname for a cmdlet or for a command element, such as a function, script, file, or executable file. You can use the alias instead of the command name in any Windows PowerShell commands.

To create an alias, use the New-Alias cmdlet. For example, the following command creates the "gas" alias for the Get-AuthenticodeSignature cmdlet:

```
New-Alias -Name gas -Value Get-AuthenticodeSignature
```

After you create the alias for the cmdlet name, you can use the alias instead of the cmdlet name. For example, to get the Authenticode signature for the SqlScript.ps1 file, type:

```
Get-AuthenticodeSignature SqlScript.ps1
```

Or, type:

```
gas SqlScript.ps1
```

If you create "word" as the alias for Microsoft Office Word, you can type "word" instead of the following:

```
"C:\Program Files\Microsoft Office\Office11\Winword.exe"
```

BUILT-IN ALIASES

Windows PowerShell includes a set of built-in aliases, including "cd" and "chdir" for the Set-Location cmdlet, and "ls" and "dir" for the Get-ChildItem cmdlet.

To get all the aliases on the computer, including the built-in aliases, type:

```
Get-Alias
```

ALIAS CMDLETS

Windows PowerShell includes the following cmdlets, which are designed for

working with aliases:

- Get-Alias. Gets all the aliases in the current session.
- New-Alias. Creates a new alias.
- Set-Alias. Creates or changes an alias.
- Export-Alias. Exports one or more aliases to a file.
- Import-Alias. Imports an alias file into Windows PowerShell.

For detailed information about the cmdlets, type:

Get-Help <cmdlet-Name> -Detailed

For example, type:

Get-Help Export-Alias -Detailed

CREATING AN ALIAS

To create a new alias, use the New-Alias cmdlet. For example, to create the "gh" alias for Get-Help, type:

New-Alias -Name gh -Value Get-Help

You can use the alias in commands, just as you would use the full cmdlet name, and you can use the alias with parameters.

For example, to get detailed Help for the Get-WmiObject cmdlet, type:

Get-Help Get-WmiObject -Detailed

Or, type:

gh Get-WmiObject -Detailed

SAVING ALIASES

The aliases that you create are saved only in the current session. To use the aliases in a different session, add the alias to your Windows PowerShell profile. Or, use the Export-Alias cmdlet to save the aliases to a file.

For more information, type:

Get-Help about_Profiles

GETTING ALIASES

To get all the aliases in the current session, including the built-in aliases, the aliases in your Windows PowerShell profiles, and the aliases that you have created in the current session, type:

Get-Alias

To get particular aliases, use the Name parameter of the Get-Alias cmdlet. For example, to get aliases that begin with "p", type:

```
Get-Alias -Name p*
```

To get the aliases for a particular item, use the Definition parameter. For example, to get the aliases for the Get-ChildItem cmdlet type:

```
Get-Alias -Definition Get-ChildItem
```

GET-ALIAS OUTPUT

Get-Alias returns only one type of object, an AliasInfo object (System.Management.Automation.AliasInfo). However, beginning in Windows PowerShell 3.0, the name of aliases that don't include a hyphen, such as "cd" are displayed in the following format:

<alias> -> <definition>

For example,

ac -> Add-Content

This makes it very quick and easy to get the information that you need.

The arrow-based alias name format is not used for aliases that include a hyphen. These are likely to be preferred substitute names for cmdlets and functions, instead of typical abbreviations or nicknames, and the author might not want them to be as evident.

ALTERNATE NAMES FOR COMMANDS WITH PARAMETERS

You can assign an alias to a cmdlet, script, function, or executable file. However, you cannot assign an alias to a command and its 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.

However, you can create a function that includes the command. To create a function, type the word "function" followed by a name for the function. Type the command, and enclose it in braces ({}).

For example, the following command creates the syslog function. This function represents the "Get-Eventlog -LogName System" command:

```
function syslog {Get-Eventlog -LogName System}
```

You can now type "syslog" instead of the command. And, you can create aliases for the syslog function.

For more information about functions, type:

```
Get-Help about_Functions
```

ALIAS OBJECTS

Windows PowerShell aliases are represented by objects that are instances of the System.Management.Automation.AliasInfo class. For more information about this type of object, see "AliasInfo Class" in the Microsoft Developer Network (MSDN) library at http://go.microsoft.com/fwlink/?LinkId=143644.

To view the properties and methods of the alias objects, get the aliases. Then, pipe them to the Get-Member cmdlet. For example:

```
Get-Alias | Get-Member
```

To view the values of the properties of a specific alias, such as the "dir" alias, get the alias. Then, pipe it to the Format-List cmdlet. For example, the following command gets the "dir" alias. Next, the command pipes the alias to the Format-List cmdlet. Then, the command uses the Property parameter of Format-List with a wildcard character (*) to display all the properties of the "dir" alias. The following command performs these tasks:

```
Get-Alias -Name dir | Format-List -Property *
```

WINDOWS POWERSHELL ALIAS PROVIDER

Windows PowerShell includes the Alias provider. The Alias provider lets you view the aliases in Windows PowerShell as though they were on a file system drive.

The Alias provider exposes the Alias: drive. To go into the Alias: drive, type:

```
Set-Location Alias:
```

To view the contents of the drive, type:

```
Get-ChildItem
```

To view the contents of the drive from another Windows PowerShell drive, begin the path with the drive name. Include the colon (:). For example:

```
Get-ChildItem -Path Alias:
```

To get information about a particular alias, type the drive name and the alias name. Or, type a name pattern. For example, to get all the aliases that begin with "p", type:

```
Get-ChildItem -Path Alias:p*
```

For more information about the Windows PowerShell Alias provider, type:

```
Get-Help Alias
```

SEE ALSO

New-Alias
Get-Alias
set-alias
export-alias
import-alias
get-psprovider
get-psdrive
about_functions
about_profiles
about_providers

about_Arithmetic_Operators

SHORT DESCRIPTION

Describes the operators that perform arithmetic in Windows PowerShell.

LONG DESCRIPTION

Arithmetic operators calculate numeric values. You can use one or more arithmetic operators to add, subtract, multiply, and divide values, and to calculate the remainder (modulus) of a division operation.

In addition, the addition operator (+) and multiplication operator (*) also operate on strings, arrays, and hash tables. The addition operator concatenates the input. The multiplication operator returns multiple copies of the input. You can even mix object types in an arithmetic statement. The method that is used to evaluate the statement is determined by the type of the leftmost object in the expression.

Beginning in Windows PowerShell 2.0, all arithmetic operators work

on 64-bit numbers.

Beginning in Windows PowerShell 3.0, the -shr (shift-right) and -shl (shift-left) are added to support bitwise arithmetic in Windows PowerShell.

Windows PowerShell supports the following arithmetic operators:

Operator	Description	Example
+	Adds integers; concatenates strings, arrays, and hash tables.	6+2 "file" + "name"
-	Subtracts one value from another value.	6-2 (get-date).date - 1
-	Makes a number a negative number.	-6+2 -4
*	Multiplies integers; copies strings and arrays the specified number of times.	6*2 "w" * 3
/	Divides two values.	6/2
%	Returns the remainder of a division operation.	7%2
-shl	Shift-left	100 -shl 2
-shr	Shift-right	100 -shr 1

OPERATOR PRECEDENCE

Windows PowerShell processes arithmetic operators in the following order:

Parentheses ()
- (for a negative number)
*, /, %
+, - (for subtraction)

Windows PowerShell processes the expressions from left to right according to the precedence rules. The following examples show the effect of the precedence rules:

```
C:\PS> 3+6/3*4
11

C:\PS> 10+4/2
12

C:\PS> (10+4)/2
7

C:\PS> (3+3)/ (1+1)
3
```

The order in which Windows PowerShell evaluates expressions might differ from other programming and scripting languages that you have used. The following example shows a complicated assignment statement.

```
C:\PS> $a = 0
C:\PS> $b = 1,2
C:\PS> $c = -1,-2

C:\PS> $b[$a] = $c[$a++]

C:\PS> $b
1
-1
```

In this example, the expression $a++ is evaluated before $c[$a++]. Evaluating $a++ changes the value of $a. The variable $a in $b[$a] equals 1, not 0, so the statement assigns a value to $b[1], not $b[0].

DIVISION AND ROUNDING

When the quotient of a division operation is an integer, Windows PowerShell rounds the value to the nearest integer. When the value is .5, it rounds to the nearest even integer.

The following example shows the effect of rounding to the nearest even integer.

```
C:\PS> [int]( 5 / 2 )
2

C:\PS> [int]( 7 / 2 )
4
```

ADDING AND MULTIPLYING NON-NUMERIC TYPES

You can add numbers, strings, arrays, and hash tables. And, you can multiply numbers, strings, and arrays. However, you cannot multiply hash tables.

When you add strings, arrays, or hash tables, the elements are concatenated. When you concatenate collections, such as arrays or hash tables, a new object is created that contains the objects from both collections. If you try to concatenate hash tables that have the same key, the operation fails.

For example, the following commands create two arrays and then add them:

```
C:\PS> $a = 1,2,3
C:\PS> $b = "A","B","C"
C:\PS> $a + $b
1
2
3
A
B
C
```

You can also perform arithmetic operations on objects of different types. The operation that Windows PowerShell performs is determined by the Microsoft .NET Framework type of the leftmost object in the operation. Windows PowerShell tries to convert all the objects in the operation to the .NET Framework type of the first
object. If it succeeds in converting the objects, it performs the operation appropriate to the .NET Framework type of the first object. If it fails to convert any of the objects, the operation fails.

The following example demonstrates the use of the addition and multiplication operators in operations that include different object types:

```
C:\PS> "file" + 16
file16

C:\PS> $array = 1,2,3
C:\PS> $array + 16
1
2
3
16
```

```
C:\PS> $array + "file"
    1
    2
    3
    file

    C:\PS> "file" * 3
    filefilefile
```

Because the method that is used to evaluate statements is determined by the leftmost object, addition and multiplication in Windows PowerShell are not strictly commutative. For example, (a + b) does not always equal (b + a), and (a * b) does not always equal (b * a).

The following examples demonstrate this principle:

```
C:\PS> "file" + 2
file2

C:\PS> 2 + "file"
Cannot convert value "file" to type "System.Int32". Error: "Input
string was not in a correct format."
At line:1 char:4
+ 2 + <<<<  "file"

C:\PS> "file" * 3
filefilefile

C:\PS> 3 * "file"
Cannot convert value "file" to type "System.Int32". Error: "Input
string was not in a correct format."
At line:1 char:4
+ 3 * <<<<  "file"
```

Hash tables are a slightly different case. You can add hash tables. And, you can add a hash table to an array. However, you cannot add any other type to a hash table.

The following examples show how to add hash tables to each other and to other objects:

```
C:\PS> $hash1 = @{a=1; b=2; c=3}
C:\PS> $hash2 = @{c1="Server01"; c2="Server02"}
C:\PS> $hash1 + $hash2
```

```
Name                           Value
----                           -----
c2                             Server02
a                              1
b                              2
c1                             Server01
c                              3


C:\PS> $hash1 + 2
You can add another hash table only to a hash table.
At line:1 char:9
+ $hash1 + <<<<  2


C:\PS> 2 + $hash1
Cannot convert "System.Collections.Hashtable" to "System.Int32".
At line:1 char:4
+ 2 + <<<<  $hash1
```

The following examples demonstrate that you can add a hash table to an array. The entire hash table is added to the array as a single object.

```
C:\PS> $array = 1,2,3
C:\PS> $array + $hash1
1
2
3

Name                           Value
----                           -----
a                              1
b                              2
c                              3


C:\PS> $sum = $array + $hash1
C:\PS> $sum.count
4

C:\PS> $sum[3]
Name                           Value
----                           -----
a                              1
b                              2
c                              3
```

```
PS C:\ps-test> $sum + $hash2
1
2
3

Name                           Value
----                           -----
a                              1
b                              2
c                              3
c2                             Server02
```

The following example shows that you cannot add hash tables that contain the same key:

```
C:\PS> $hash1 = @{a=1; b=2; c=3}
C:\PS> $hash2 = @{c="red"}
C:\PS> $hash1 + $hash2
Bad argument to operator '+': Item has already been added.
Key in dictionary: 'c'    Key being added: 'c'.
At line:1 char:9
+ $hash1 + <<<<  $hash2
```

Although the addition operators are very useful, use the assignment operators to add elements to hash tables and arrays. For more information see about_assignment_operators. The following examples use the += assignment operator to add items to an array:

```
C:\PS>  $array
1
2
3

C:\PS>  $array + "file"
1
2
3
file

C:\PS>  $array
1
2
3
```

```
C:\PS>  $array += "file"
C:\PS>  $array
1
2
3
file

C:\PS> $hash1

Name                   Value
----                   -----
a                      1
b                      2
c                      3

C:\PS> $hash1 += @{e = 5}
C:\PS> $hash1

Name                   Value
----                   -----
a                      1
b                      2
e                      5
c                      3
```

Windows PowerShell automatically selects the .NET Framework numeric type that best expresses the result without losing precision. For example:

```
C:\PS> 2 + 3.1
5.1
C:\PS> (2). GetType().FullName
System.Int32
C:\PS> (2 + 3.1).GetType().FullName
System.Double
```

If the result of an operation is too large for the type, the type of the result is widened to accommodate the result, as in the following example:

```
C:\PS> (512MB).GetType().FullName
System.Int32
C:\PS> (512MB * 512MB).GetType().FullName
System.Double
```

The type of the result will not necessarily be the same as one of the operands. In the following example, the negative value cannot be cast to an unsigned integer, and the unsigned integer is too large to be cast to

Int32:

```
C:\PS> ([int32]::minvalue + [uint32]::maxvalue).gettype().fullname
System.Int64
```

In this example, Int64 can accommodate both types.

The System.Decimal type is an exception. If either operand has the Decimal type, the result will be of the Decimal type. If the result is too large for the Decimal type, it will not be cast to Double. Instead, an error results.

```
C:\PS> [Decimal]::maxvalue
79228162514264337593543950335
C:\PS> [Decimal]::maxvalue + 1
Value was either too large or too small for a Decimal.
At line:1 char:22
+ [Decimal]::maxvalue + <<<<  1
```

ARITHMETIC OPERATORS AND VARIABLES

You can also use arithmetic operators with variables. The operators act on the values of the variables. The following examples demonstrate the use of arithmetic operators with variables:

```
C:\PS> $intA = 6
C:\PS> $intB = 4
C:\PS> $intA + $intB

10
```

```
C:\PS> $a = "Windows "
C:\PS> $b = "PowerShell "
C:\PS> $c = 2
C:\PS> $a + $b + $c

Windows PowerShell 2
```

ARITHMETIC OPERATORS AND COMMANDS

Typically, you use the arithmetic operators in expressions with numbers, strings, and arrays. However, you can also use arithmetic operators with the objects that commands return and with the properties of those objects.

The following examples show how to use the arithmetic operators in expressions with Windows PowerShell commands:

```
C:\PS> get-date
Wednesday, January 02, 2008 1:28:42 PM


C:\PS> $day = new-timespan -day 1
C:\PS> get-date + $day
Thursday, January 03, 2008 1:34:52 PM


C:\PS> get-process | where {($_.ws * 2) -gt 50mb}
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1896      39    50968      30620   264 1,572.55   1104 explorer
  12802      78   188468      81032   753 3,676.39   5676 OUTLOOK
    660       9    36168      26956   143    12.20    988 PowerShell
    561      14     6592      28144   110 1,010.09    496 services
   3476      80    34664      26092   234 ...45.69    876 svchost
    967      30    58804      59496   416   930.97   2508 WINWORD


 EXAMPLES
    The following examples show how to use the arithmetic operators in
    Windows PowerShell:

C:\PS> 1 + 1
2

C:\PS> 1 - 1
0

C:\PS> -(6 + 3)
-9

C:\PS> 6 * 2
12

C:\PS> 7 / 2
3.5

C:\PS> 7 % 2
1

C:\PS> w * 3
www

C:\PS> 3 * "w"
Cannot convert value "w" to type "System.Int32". Error: "Input string was not
```

```
      in a correct format."
At line:1 char:4
+ 3 * <<<< "w"

PS C:\ps-test> "Windows" + " " + "PowerShell"
Windows PowerShell

PS C:\ps-test> $a = "Windows" + " " + "PowerShell"
PS C:\ps-test> $a
Windows PowerShell

C:\PS> $a[0]
W

C:\PS> $a = "TestFiles.txt"
C:\PS> $b = "C:\Logs\"
C:\PS> $b + $a
C:\Logs\TestFiles.txt

C:\PS> $a = 1,2,3
C:\PS> $a + 4
1
2
3
4

C:\PS> $servers = @{0 = "LocalHost"; 1 = "Server01"; 2 = "Server02"}
C:\PS> $servers + @{3 = "Server03"}
Name Value
---- -----
3 Server03
2 Server02
1 Server01
0 LocalHost

C:\PS> $servers
Name Value
---- -----
2 Server02
1 Server01
0 LocalHost

C:\PS> $servers += @{3 = "Server03"} #Use assignment operator
C:\PS> $servers
Name Value
---- -----
3 Server03
2 Server02
```

1 Server01
0 LocalHost

BITWISE ARITHMETIC IN WINDOWS POWERSHELL

Windows PowerShell supports the -shl (shift-left) and
-shr (shift-right) operators for bitwise arithmetic.

These operators are introduced in Windows
PowerShell 3.0.

In a bitwise shift-left operation, all bits are moved "n" places to
the left, where "n" is the value of the right operand. A zero is
inserted in the ones place.

When the left operand is an Integer (32-bit) value, the lower 5 bits
of the right operand determine how many bits of the left operand are
shifted.

When the left operand is a Long (64-bit) value, the lower 6 bits of
the right operand determine how many bits of the left operand are
shifted.

```
PS C:\> 21 -shl 1
42

 00010101  (21)
 00101010  (42)

PS C:\> 21 -shl 2
84

 00010101  (21)
 00101010  (42)
 01010100  (84)
```

In a bitwise shift-right operation, all bits are moved "n" places
to the right, where "n" is specified by the right operand. The
shift-right operator (-shr) inserts a zero in the left-most
place when shifting a positive or unsigned value to the right.

When the left operand is an Integer (32-bit) value, the lower 5 bits
of the right operand determine how many bits of the left operand are
shifted.

When the left operand is a Long (64-bit) value, the lower 6 bits of the right operand determine how many bits of the left operand are shifted.

```
PS C:\> 21 -shr 1
10

 00010101  (21)
 00001010  (10)

PS C:\> 21 -shr 2
5

 00010101  (21)
 00001010  (10)
 00000101  ( 5)
```

SEE ALSO

about_arrays
about_assignment_operators
about_comparison_operators
about_hash_tables
about_operators
about_variables
Get-Date
New-TimeSpan

about_Arrays

SHORT DESCRIPTION

Describes arrays, which are data structures designed to store collections of items.

LONG DESCRIPTION

An array is a data structure that is designed to store a collection of items. The items can be the same type or different types.

Beginning in Windows PowerShell 3.0, a collection of zero or one object has some properties of arrays.

CREATING AND INITIALIZING AN ARRAY

To create and initialize an array, assign multiple values to a variable.

The values stored in the array are delimited with a comma and separated from the variable name by the assignment operator (=).

For example, to create an array named $A that contains the seven numeric (int) values of 22, 5, 10, 8, 12, 9, and 80, type:

```
$A = 22,5,10,8,12,9,80
```

You can also create and initialize an array by using the range operator (..). For example, to create and initialize an array named "$B" that contains the values 5 through 8, type:

```
$B = 5..8
```

As a result, $B contains four values: 5, 6, 7, and 8.

When no data type is specified, Windows PowerShell creates each array as an object array (type: System.Object[]). To determine the data type of an array, use the GetType() method. For example, to determine the data type of the $a array, type:

```
$a.GetType()
```

To create a strongly typed array, that is, an array that can contain only values of a particular type, cast the variable as an array type, such as string[], long[], or int32[]. To cast an array, precede the variable name with an array type enclosed in brackets. For example, to create a 32-bit integer array named $ia containing four integers (1500, 2230, 3350, and 4000), type:

```
[int32[]]$ia = 1500,2230,3350,4000
```

As a result, the $ia array can contain only integers.

You can create arrays that are cast to any supported type in the Microsoft .NET Framework. For example, the objects that Get-Process retrieves to represent processes are of the System.Diagnostics.Process type. To create a strongly typed array of process objects, enter the following command:

```
[Diagnostics.Process[]]$zz = Get-Process
```

THE ARRAY SUB-EXPRESSION OPERATOR

The array sub-expression operator creates an array, even if it contains zero or one object.

The syntax of the array operator is as follows:

```
@( ... )
```

You can use the array operator to create an array of zero or one object.

```
PS C:\>$a = @("One")
PS C:\>$a.Count
1

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

The array operator is particularly useful in scripts when you are getting objects, but do not know how many objects you will get.

```
PS C:\> $p = @(Get-Process Notepad)
```

For more information about the array sub-expression operator, see about_Operators.

READING AN ARRAY

You can refer to an array by using its variable name. To display all the elements in the array, type the array name. For example:

```
$a
```

You can refer to the elements in an array by using an index, beginning at position 0. Enclose the index number in brackets. For example, to display the first element in the $a array, type:

```
$a[0]
```

To display the third element in the $a array, type:

```
$a[2]
```

Negative numbers count from the end of the array. For example, "-1" refers to the last element of the array. To display the last three elements of the array, type:

```
$a[-3..-1]
```

However, be cautious when using this notation.

```
$a[0..-2]
```

This command does not refer to all the elements of the array, except for the last one. It refers to the first, last, and second-to-last elements in the array.

You can use the range operator to display a subset of all the values in an array. For example, to display the data elements at index position 1 through 3, type:

```
$a[1..3]
```

You can use the plus operator (+) to combine a range with a list of elements in an array. For example, to display the elements at index positions 0, 2, and 4 through 6, type:

```
$a[0,2+4..6]
```

To determine how many items are in an array, use the Length property or its Count alias.

```
$a.Count
```

You can also use looping constructs, such as ForEach, For, and While loops, to refer to the elements in an array. For example, to use a ForEach loop to display the elements in the $a array, type:

```
foreach ($element in $a) {$element}
```

The Foreach loop iterates through the array and returns each value in the array until reaching the end of the array.

The For loop is useful when you are incrementing counters while examining the elements in an array. For example, to use a For loop to return every other value in an array, type:

```
for ($i = 0; $i -le ($a.length - 1); $i += 2) {$a[$i]}
```

You can use a While loop to display the elements in an array until a defined condition is no longer true. For example, to display the elements in the $a array while the array index is less than 4, type:

```
$i=0
while($i -lt 4) {$a[$i]; $i++}
```

GET THE MEMBERS OF AN ARRAY

To get the properties and methods of an array, such as the Length property and the SetValue method, use the InputObject parameter of the Get-Member cmdlet.

When you pipe an array to Get-Member, Windows PowerShell sends the items one at a time and Get-Member returns the type of each item in the array (ignoring duplicates).

When you use the InputObject parameter, Get-Member returns the members of the array.

For example, the following command gets the members of the array in the $a variable.

```
Get-Member -InputObject $a
```

You can also get the members of an array by typing a comma (,) before the value that is piped to the Get-Member cmdlet. The comma makes the array the second item in an array of arrays. Windows PowerShell pipes the arrays one at a time and Get-Member returns the members of the array.

```
,$a | Get-Member

,(1,2,3) | Get-Member
```

MANIPULATING AN ARRAY

You can change the elements in an array, add an element to an array, and combine the values from two arrays into a third array.

To change the value of a particular element in an array, specify the array name and the index of the element that you want to change, and then use the assignment operator (=) to specify a new value for the element. For example, to change the value of the second item in the $a array (index position 1) to 10, type:

```
$a[1] = 10
```

You can also use the SetValue method of an array to change a value. The following example changes the second value (index position 1) of the $a array to 500:

```
$a.SetValue(500,1)
```

You can use the += operator to add an element to an array. When you use it, Windows PowerShell actually creates a new array with the values of the original array and the added value. For example, to add an element with a

value of 200 to the array in the $a variable, type:

```
$a += 200
```

It is not easy to delete elements from an array, but you can create a new array that contains only selected elements of an existing array. For example, to create the $t array with all the elements in the $a array except for the value at index position 2, type:

```
$t = $a[0,1 + 3..($a.length - 1)]
```

To combine two arrays into a single array, use the plus operator (+). The following example creates two arrays, combines them, and then displays the resulting combined array.

```
$x = 1,3
$y = 5,9
$z = $x + $y
```

As a result, the $z array contains 1, 3, 5, and 9.

To delete an array, assign a value of $null to the array. The following command deletes the array in the $a variable.

```
$a = $null
```

You can also use the Remove-Item cmdlet, but assigning a value of $null is faster, especially for large arrays.

ARRAYS OF ZERO OR ONE

Beginning in Windows PowerShell 3.0, a collection of zero or one object has the Count and Length property. Also, you can index into an array of one object. This feature helps you to avoid scripting errors that occur when a command that expects a collection gets fewer than two items.

The following examples demonstrate this feature.

```
#Zero objects
$a = $null
$a.Count
0
$a.Length
0

#One object
```

```
$a = 4
$a.Count
1
$a.Length
1
$a[0]
4
$a[-1]
4
```

SEE ALSO

about_Assignment_Operators
about_Hash_Tables
about_Operators
about_For
about_Foreach
about_While

about_assignment_operators

SHORT DESCRIPTION

Describes how to use operators to assign values to variables.

LONG DESCRIPTION

Assignment operators assign one or more values to a variable. They can perform numeric operations on the values before the assignment.

Windows PowerShell supports the following assignment operators.

Operator	Description
=	Sets the value of a variable to the specified value.
+=	Increases the value of a variable by the specified value, or appends the specified value to the existing value.

-= Decreases the value of a variable by the specified value.

*= Multiplies the value of a variable by the specified value, or appends the specified value to the existing value.

/= Divides the value of a variable by the specified value.

%= Divides the value of a variable by the specified value and then assigns the remainder (modulus) to the variable.

++ Increases the value of a variable, assignable property, or array element by 1.

-- Decreases the value of a variable, assignable property, or array element by 1.

SYNTAX

The syntax of the assignment operators is as follows:

<assignable-expression> <assignment-operator> <value>

Assignable expressions include variables and properties. The value can be a single value, an array of values, or a command, expression, or statement.

The increment and decrement operators are unary operators. Each has prefix and postfix versions.

<assignable-expression><operator>
<operator><assignable-expression>

The assignable expression must a number or it must be convertible to a number.

ASSIGNING VALUES

Variables are named memory spaces that store values. You store the values in variables by using the assignment operator (=). The new value can replace the existing value of the variable, or you can append a new value to the existing value.

The basic assignment operator is the equal sign (=)(ASCII 61). For example, the following statement assigns the value Windows PowerShell to the $MyShell variable:

```
$MyShell = "Windows PowerShell"
```

When you assign a value to a variable in Windows PowerShell, the variable is created if it did not already exist. For example, the first of the following two assignement statements creates the $a variable and assigns a value of 6 to $a. The second assignment statement assigns a value of 12 to $a. The first statement creates a new variable. The second statement changes only its value:

```
$a = 6
$a = 12
```

Variables in Windows PowerShell do not have a specific data type unless you cast them. When a variable contains only one object, the variable takes the data type of that object. When a variable contains a collection of objects, the variable has the System.Object data type. Therefore, you can assign any type of object to the collection. The following example shows that you can add process objects, service objects, strings, and integers to a variable without generating an error:

```
$a = get-process
$a += get-service
$a += "string"
$a += 12
```

Because the assignment operator (=) has a lower precedence than the pipeline operator (|), parentheses are not required to assign the result of a command pipeline to a variable. For example, the following command sorts the services on the computer and then assigns the sorted services to the $a variable:

```
$a = get-service | sort name
```

You can also assign the value created by a statement to a variable, as in the following example:

```
$a = if ($b -lt 0) { 0 } else { $b }
```

This example assigns 0 to the $a variable if the value of $b is less than 0. It assigns the value of $b to $a if the value of $b is not less than zero.

THE ASSIGNMENT OPERATOR (=)

The assignment operator (=) assigns values to variables. If the variable already has a value, the assignment operator (=) replaces the value without warning.

The following statement assigns the integer value 6 to the $a variable:

```
$a = 6
```

To assign a string value to a variable, enclose the string value in quotation marks, as follows:

```
$a = "baseball"
```

To assign an array (multiple values) to a variable, separate the values with commas, as follows:

```
$a = "apple", "orange", "lemon", "grape"
```

To assign a hash table to a variable, use the standard hash table notation in Windows PowerShell. Type an at sign (@) followed by key/value pairs that are separated by semicolons (;) and enclosed in braces ({ }). For example, to assign a hash table to the $a variable, type:

```
$a = @{one=1; two=2; three=3}
```

To assign hexadecimal values to a variable, precede the value with "0x". Windows PowerShell converts the hexadecimal value (0x10) to a decimal value (in this case, 16) and assigns that value to the $a variable. For

example, to assign a value of 0x10 to the $a variable, type:

```
$a = 0x10
```

To assign an exponential value to a variable, type the root number, the letter "e", and a number that represents a multiple of 10. For example, to assign a value of 3.1415 to the power of 1,000 to the $a variable, type:

```
$a = 3.1415e3
```

Windows PowerShell can also convert kilobytes (KB), megabytes (MB), and gigabytes (GB) into bytes. For example, to assign a value of 10 kilobytes to the $a variable, type:

```
$a = 10kb
```

THE ASSIGNMENT BY ADDITION OPERATOR (+=)

The assignment by addition operator (+=) either increments the value of a variable or appends the specified value to the existing value. The action depends on whether the variable has a numeric or string type and whether the variable contains a single value (a scalar) or multiple values (a collection).

The += operator combines two operations. First, it adds, and then it assigns. Therefore, the following statements are equivalent:

```
$a += 2
$a = ($a + 2)
```

When the variable contains a single numeric value, the += operator increments the existing value by the amount on the right side of the operator. Then, the operator assigns the resulting value to the variable. The following example shows how to use the += operator to increase the value of a variable:

```
C:\PS> $a = 4
C:\PS> $a += 2
C:\PS> $a
```

When the value of the variable is a string, the value on the right side of the operator is appended to the string, as follows:

```
C:\PS> $a = "Windows"
C:\PS> $a += " PowerShell"
C:\PS> $a
Windows PowerShell
```

When the value of the variable is an array, the += operator appends the values on the right side of the operator to the array. Unless the array is explicitly typed by casting, you can append any type of value to the array, as follows:

```
C:\PS> $a = 1,2,3
C:\PS> $a += 2
C:\PS> $a
1
2
3
2
C:\PS> $a += "String"
C:\PS> $a
1
2
3
2
String
```

When the value of a variable is a hash table, the += operator appends the value on the right side of the operator to the hash table. However, because the only type that you can add to a hash table is another hash table, all other assignments fail.

For example, the following command assigns a hash table to the $a variable. Then, it uses the += operator to append another hash table to the existing hash table, effectively adding a new key/value pair to the existing hash table. This command succeeds, as shown in the output:

```
C:\PS> $a = @{a = 1; b = 2; c = 3}
```

```
C:\PS> $a += @{mode = "write"}
C:\PS> $a
Name                           Value
----                           -----
a                              1
b                              2
mode                           write
c                              3
```

The following command attempts to append an integer (1) to the hash table in the $a variable. This command fails:

```
C:\PS> $a = @{a = 1; b = 2; c = 3}
C:\PS> $a += 1
You can add another hash table only to a hash table.
At line:1 char:6
+ $a += <<<<  1
```

THE ASSIGNMENT BY SUBTRACTION OPERATOR (-=)

The assignment by subtraction operator (-=) decrements the value of a variable by the value that is specified on the right side of the operator. This operator cannot be used with string variables, and it cannot be used to remove an element from a collection.

The -= operator combines two operations. First, it subtracts, and then it assigns. Therefore, the following statements are equivalent:

```
$a -= 2
$a = ($a - 2)
```

The following example shows how to use of the -= operator to decrease the value of a variable:

```
C:\PS> $a = 8
C:\PS> $a -= 2
C:\PS> $a
6
```

You can also use the -= assignment operator to decrease the value of a member of a numeric array. To do this, specify the index of the array

element that you want to change. In the following example, the value of the third element of an array (element 2) is decreased by 1:

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

You cannot use the -= operator to delete the values of a variable. To delete all the values that are assigned to a variable, use the Clear-Item or Clear-Variable cmdlets to assign a value of $null or "" to the variable.

```
$a = $null
```

To delete a particular value from an array, use array notation to assign a value of $null to the particular item. For example, the following statement deletes the second value (index position 1) from an array:

```
C:\PS> $a = 1,2,3
C:\PS> $a
1
2
3

C:\PS> $a[1] = $null
C:\PS> $a
1
3
```

To delete a variable, use the Remove-Variable cmdlet. This method is useful when the variable is explicitly cast to a particular data type, and you want an untyped variable. The following command deletes the $a variable:

```
remove-variable a
```

THE ASSIGNMENT BY MULTIPLICATION OPERATOR (*=)

The assignment by multiplication operator (*=) multiplies a numeric value

or appends the specified number of copies of the string value of a variable.

When a variable contains a single numeric value, that value is multiplied by the value on the right side of the operator. For example, the following example shows how to use the *= operator to multiply the value of a variable:

```
C:\PS> $a = 3
C:\PS> $a *= 4
C:\PS> $a
12
```

In this case, the *= operator combines two operations. First, it multiplies, and then it assigns. Therefore, the following statements are equivalent:

```
$a *= 2
$a = ($a * 2)
```

When a variable contains a string value, Windows PowerShell appends the specified number of strings to the value, as follows:

```
C:\PS> $a = "file"
C:\PS> $a *= 4
C:\PS> $a
filefilefilefile
```

To multiply an element of an array, use an index to identify the element that you want to multiply. For example, the following command multiplies the first element in the array (index position 0) by 2:

```
$a[0] *= 2
```

THE ASSIGNMENT BY DIVISION OPERATOR (/=)

The assignment by division operator (/=) divides a numeric value by the value that is specified on the right side of the operator. The operator cannot be used with string variables.

The /= operator combines two operations. First, it divides, and then it assigns. Therefore, the following two statements are equivalent:

```
$a /= 2
$a = ($a / 2)
```

For example, the following command uses the /= operator to divide the value of a variable:

```
C:\PS> $a = 8
C:\PS> $a /=2
C:\PS> $a
4
```

To divide an element of an array, use an index to identify the element that you want to change. For example, the following command divides the second element in the array (index position 1) by 2:

```
$a[1] /= 2
```

THE ASSIGNMENT BY MODULUS OPERATOR (%=)

The assignment by modulus operator (%=) divides the value of a variable by the value on the right side of the operator. Then, the %= operator assigns the remainder (known as the modulus) to the variable. You can use this operator only when a variable contains a single numeric value. You cannot use this operator when a variable contains a string variable or an array.

The %= operator combines two operations. First, it divides and determines the remainder, and then it assigns the remainder to the variable. Therefore, the following statements are equivalent:

```
$a %= 2
$a = ($a % 2)
```

The following example shows how to use the %= operator to save the modulus of a quotient:

```
C:\PS> $a = 7
C:\PS> $a %= 4
C:\PS> $a
3
```

THE INCREMENT AND DECREMENT OPERATORS

The increment operator (++) increases the value of a variable by 1. When you use the increment operator in a simple statement, no value is returned. To view the result, display the value of the variable, as follows:

```
C:\PS> $a = 7
C:\PS> ++$a
C:\PS> $a
8
```

To force a value to be returned, enclose the variable and the operator in parentheses, as follows:

```
C:\PS> $a = 7
C:\PS> (++$a)
8
```

The increment operator can be placed before (prefix) or after (postfix) a variable. The prefix version of the operator increments a variable before its value is used in the statement, as follows:

```
C:\PS> $a = 7
C:\PS> $c = ++$a
C:\PS> $a
8
C:\PS> $c
8
```

The postfix version of the operator increments a variable after its value is used in the statement. In the following example, the $c and $a variables have different values because the value is assigned to $c before $a changes:

```
C:\PS> $a = 7
```

```
C:\PS> $c = $a++
C:\PS> $a
8
C:\PS> $c
7
```

The decrement operator (--) decreases the value of a variable by 1. As with the increment operator, no value is returned when you use the operator in a simple statement. Use parentheses to return a value, as follows:

```
C:\PS> $a = 7
C:\PS> --$a
C:\PS> $a
6
C:\PS> (--$a)
5
```

The prefix version of the operator decrements a variable before its value is used in the statement, as follows:

```
C:\PS> $a = 7
C:\PS> $c = --$a
C:\PS> $a
6
C:\PS> $c
6
```

The postfix version of the operator decrements a variable after its value is used in the statement. In the following example, the $d and $a variables have different values because the value is assigned to $d before $a changes:

```
C:\PS> $a = 7
C:\PS> $d = $a--
C:\PS> $a
6
C:\PS> $d
7
```

MICROSOFT .NET FRAMEWORK TYPES

By default, when a variable has only one value, the value that is assigned

to the variable determines the data type of the variable. For example, the following command creates a variable that has the Integer (System.Int32) type:

```
$a = 6
```

To find the .NET Framework type of a variable, use the GetType method and its FullName property, as follows. Be sure to include the parentheses after the GetType method name, even though the method call has no arguments:

```
C:\PS> $a = 6
C:\PS> $a.gettype().fullname
System.Int32
```

To create a variable that contains a string, assign a string value to the variable. To indicate that the value is a string, enclose it in quotation marks, as follows:

```
C:\PS> $a = "6"
C:\PS> $a.gettype().fullname
System.String
```

If the first value that is assigned to the variable is a string, Windows PowerShell treats all operations as string operations and casts new values to strings. This occurs in the following example:

```
C:\PS> $a = "file"
C:\PS> $a += 3
C:\PS> $a
file3
```

If the first value is an integer, Windows PowerShell treats all operations as integer operations and casts new values to integers. This occurs in the following example:

```
C:\PS> $a = 6
C:\PS> $a += "3"
C:\PS> $a
9
```

You can cast a new scalar variable as any .NET Framework type by placing the type name in brackets that precede either the variable name or the first assignment value. When you cast a variable, you can determine the types of data that can be stored in the variable. And, you can determine how the variable behaves when you manipulate it.

For example, the following command casts the variable as a string type:

```
C:\PS> [string]$a = 27
C:\PS> $a += 3
C:\PS> $a
273
```

The following example casts the first value, instead of casting the variable:

```
$a = [string]27
```

When you cast a variable to a specific type, the common convention is to cast the variable, not the value. However, you cannot recast the data type of an existing variable if its value cannot be converted to the new data type. To change the data type, you must replace its value, as follows:

```
C:\PS> $a = "string"
C:\PS> [int]$a
Cannot convert value "string" to type "System.Int32". Error: "Input
string was not in a correct format."
At line:1 char:8
+ [int]$a <<<<

C:\PS> [int]$a =3
```

In addition, when you precede a variable name with a data type, the type of that variable is locked unless you explicitly override the type by specifying another data type. If you try to assign a value that is incompatible with the existing type, and you do not explicitly override the type, Windows PowerShell displays an error, as shown in the following example:

```
C:\PS> $a = 3
C:\PS> $a = "string"

C:\PS> [int]$a = 3
C:\PS> $a = "string"
Cannot convert value "string" to type "System.Int32". Error: "Input
string was not in a correct format."
At line:1 char:3
+ $a <<<<  = "string"

C:\PS> [string]$a = "string"
```

In Windows PowerShell, the data types of variables that contain multiple items in an array are handled differently from the data types of variables that contain a single item. Unless a data type is specifically assigned to an array variable, the data type is always System.Object []. This data type is specific to arrays.

Sometimes, you can override the default type by specifying another type. For example, the following command casts the variable as a string [] array type:

```
[string []] $a = "one", "two", "three"
```

Windows PowerShell variables can be any .NET Framework data type. In addition, you can assign any fully qualified .NET Framework data type that is available in the current process. For example, the following command specifies a System.DateTime data type:

```
[system.datetime]$a = "5/31/2005"
```

The variable will be assigned a value that conforms to the System.DateTime data type. The value of the $a variable would be the following:

```
Tuesday, May 31, 2005 12:00:00 AM
```

ASSIGNING MULTIPLE VARIABLES

In Windows PowerShell, you can assign values to multiple variables by using

a single command. The first element of the assignment value is assigned to the first variable, the second element is assigned to the second variable, the third element to the third variable, and so on. For example, the following command assigns the value 1 to the $a variable, the value 2 to the $b variable, and the value 3 to the $c variable:

```
C:\PS> $a, $b, $c = 1, 2, 3
```

If the assignment value contains more elements than variables, all the remaining values are assigned to the last variable. For example, the following command contains three variables and five values:

```
$a, $b, $c = 1, 2, 3, 4, 5
```

Therefore, Windows PowerShell assigns the value 1 to the $a variable and the value 2 to the $b variable. It assigns the values 3, 4, and 5 to the $c variable. To assign the values in the $c variable to three other variables, use the following format:

```
$d, $e, $f = $c
```

This command assigns the value 3 to the $d variable, the value 4 to the $e variable, and the value 5 to the $f variable.

You can also assign a single value to multiple variables by chaining the variables. For example, the following command assigns a value of "three" to all four variables:

```
$a = $b = $c = $d = "three"
```

VARIABLE-RELATED CMDLETS

In addition to using an assignment operation to set a variable value, you can also use the Set-Variable cmdlet. For example, the following command uses Set-Variable to assign an array of 1, 2, 3 to the $a variable.

```
Set-Variable -name a -value 1, 2, 3
```

SEE ALSO

about_Arrays
about_Hash_Tables
about_Variables
Clear-Variable
Remove-Variable
Set-Variable

about_Automatic_Variables

SHORT DESCRIPTION

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

LONG DESCRIPTION

Here is a list of the automatic variables in Windows PowerShell:

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

$?
Contains the execution status of the last operation. It contains TRUE if the last operation succeeded and FALSE if it failed.

$^
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 or on selected objects in a pipeline.

$AllNodes
This variable is available inside of a DSC configuration document when configuration data has been passed into it by using the -ConfigurationData parameter. For more information about configuration data, see "Separating Configuration and Environment Data" on Microsoft TechNet (http://technet.microsoft.com/library/dn249925.aspx).

$Args
Contains an array of the undeclared parameters and/or parameter values that are passed to a function, script, or script block.
When you create a function, you can declare the parameters by using 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 is 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 (System.Management.Automation.PSEventArgs) 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 Windows 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 which file will be updated.

$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 (http://go.microsoft.com/fwlink/?LinkID=113216).

$Event

Contains a PSEventArgs object that represents the event that is 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. Therefore, 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 is 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 (System.Management.Automation.PSEventArgs) object that Get-Event returns.

$EventSubscriber

Contains a PSEventSubscriber object that represents the event subscriber of the event that is 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 Windows 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 is converted to a non-empty string or to a non-zero integer.

$ForEach
Contains the enumerator (not the resulting values) of a ForEach loop. You can use the properties and methods of enumerators on the value of the $ForEach variable. This variable exists only while the ForEach loop is running; it is deleted after the loop is completed. For detailed information, see about_ForEach.

$Home
Contains the full path of the user's home directory. This variable is the equivalent of the %homedrive%%homepath% environment variables, typically C:\Users\<UserName>.

$Host
Contains an object that represents the current host application for Windows 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.setbackgroundcolor("Red").

$Input
Contains an enumerator that enumerates all input that is passed to a function. The $input variable is available only to functions and script blocks (which are unnamed functions). In the Process block of a function, the $input variable enumerates the object that is currently in the pipeline. When the Process block completes, there are no objects left in the pipeline, so the $input variable enumerates an empty collection. If the function does not have a Process block, then in the End block, the $input variable enumerates the collection of all input to the function.

$LastExitCode
Contains the exit code of the last Windows-based program that was run.

$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. For more information about the -match operator, see about_comparison_operators.

$MyInvocation

Contains an 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 $MyInvocation returns in the current script, such as the path and file name of the script ($MyInvocation.MyCommand.Path) or the name of a function ($MyInvocation.MyCommand.Name) to identify the current command. This is particularly useful for finding the name of the current script.

Beginning in Windows 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 file name 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, Windows PowerShell presents a nested command prompt when you use the $Host.EnterNestedPrompt method. Windows PowerShell also presents a nested command prompt when you reach a breakpoint in the Windows PowerShell debugger.

When you enter a nested prompt, Windows 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 Windows PowerShell command prompt that includes this value so that it is 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.

Windows PowerShell treats $null as an object with a value, that is, as an explicit placeholder, so you can use $null to represent an empty value in a series of values.

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

```
C:\PS> $a = ".dir", $null, ".pdf"
C:\PS> $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.

```
PS C:\ps-test> ".dir", "$null, ".pdf" | Foreach {"Hello"}
Hello
Hello
Hello
```

As a result, you cannot use $null to mean "no parameter value." A parameter value of $null overrides the default parameter value.

However, because Windows PowerShell treats the $null variable as a placeholder, you can use it scripts like the following one, which would not 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
```

$OFS

$OFS is a special variable that stores a string that you want to use as an output field separator. Use this variable when you are converting an array to a string. By default, the value of $OFS is " ", but you can change the value of $OFS in your session, by typing $OFS="<value>". If you are expecting the default value of " " in your script, module, or configuration output, be careful that the $OFS default value has not been changed elsewhere in your code.

Examples:

```
PS> $a="1","2","3","4"
PS> $a
1
2
3
4

PS> [string]$a
1 2 3 4
PS> $OFS="";[string]$a
1234

PS> $OFS=",";[string]$a
1,2,3,4

PS> $OFS="--PowerShellRocks--";[string]$a
1--PowerShellRocks--2--PowerShellRocks--3--PowerShellRocks--4
PS> $OFS="`n`n";[string]$a
1

2

3

4
```

$PID

Contains the process identifier (PID) of the process that is hosting the current Windows PowerShell session.

$Profile

Contains the full path of the Windows 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 -type file -path $pshome -force
```

You can also use it in a command to open the profile in Notepad:

```
notepad $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.

For example:

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

  # Display the parameters in dictionary format.
  $PSBoundParameters

  # Call the Test1 function with $a and $b.
  test1 @PSBoundParameters
}
```

$PsCmdlet

Contains an object that represents the cmdlet or advanced function that is 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 is 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_Advanced.

$PSCommandPath
Contains the full path and file name of the script that is being run. This variable is valid in all scripts.

$PsCulture
Contains the name of the culture currently in use in the operating system. The culture determines the display format of items such as numbers, currrency, and dates. This is the value of the System.Globalization.CultureInfo.CurrentCulture.Name property of the system. To get the System.Globalization.CultureInfo object for the system, use the Get-Culture cmdlet.

$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 indicate 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 is being debugged.

$PsHome
Contains the full path of the installation directory for Windows PowerShell, typically, %windir%\System32\WindowsPowerShell\v1.0. You can use this variable in the paths of Windows PowerShell files. For example, the following command searches the conceptual Help topics for the word "variable":

```
Select-String -Pattern Variable -Path $pshome\*.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 or on selected objects in a pipeline.

$PSScriptRoot
Contains the directory from which a script is being run.

In Windows PowerShell 2.0, this variable is valid only in script modules (.psm1). Beginning in Windows PowerShell 3.0, it is 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, which, 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 is currently in use 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 Windows PowerShell that is running in the current session. The table includes the following items:

CLRVersion: The version of the common language runtime (CLR)

BuildVersion: The build number of the current version

PSVersion: The Windows PowerShell version number

WSManStackVersion: The version number of the WS-Management stack

PSCompatibleVersions: Versions of Windows PowerShell that are compatible with the current version

SerializationVersion The version of the serialization method

PSRemotingProtocolVersion
The version of the Windows PowerShell remote management protocol

$Pwd

Contains a path object that represents the full path of the current directory.

$ReportErrorShowExceptionClass
$ReportErrorShowInnerException
$ReportErrorShowSource
$ReportErrorShowStackTrace
The "ReportErrorShow" variables are defined in Windows PowerShell, but they are not implemented. Get-Variable gets them, but they do not contain valid data.

$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 (System.Management.Automation.PSEventArgs) object that Get-Event returns.

$ShellID
Contains the identifier of the current shell.

$StackTrace
Contains a stack trace for the most recent error.

$This
In a script block that defines a script property or script method, the $This variable refers to the object that is being extended.

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

SEE ALSO
about_Hash_Tables
about_Preference_Variables
about_Variables

about_Break

SHORT DESCRIPTION
Describes a statement you can use to immediately exit Foreach, For, While, Do, or Switch statements.

LONG DESCRIPTION

When a Break statement appears in a loop, such as a Foreach, For, Switch, or While loop, the Break statement causes Windows PowerShell to immediately exit the loop. In a Switch construct that does not loop, Break causes Windows PowerShell to exit the Switch code block.

A Break statement can include a label that lets you exit embedded loops. A label can specify any loop keyword, such as Foreach, For, or While, in a script. When you use a label, Break exits the specified loop. Break exits the specified loop, regardless of which loop the Break statement is in.

The following example shows how to use a Break statement to exit a For statement:

```
for($i=1; $i -le 10; $i++)
{
   Write-Host $i
   break
}
```

In this example, the Break statement exits the For loop when the $i variable equals 1. Even though the For statement evaluates to True until $i is greater than 10, Windows PowerShell reaches the break statement the first time the For loop is run.

It is more common to use the Break statement in a loop where an inner condition must be met. Consider the following Foreach statement example:

```
$i=0
$varB = 10,20,30,40
foreach ($val in $varB)
{
   $i++
   if ($val -eq 30)
   {
      break
   }
}
Write-Host "30 was found in array position $i"
```

In this example, the Foreach statement iterates the $varB array. Each

time the code block is run, the $i variable is incremented by 1. The If statement evaluates to False the first two times the loop is run. The third time the loop is run, $i equals 3, and the $val variable equals 30. At this point, the Break statement runs, and the Foreach loop exits.

You break out of the other looping statements in the same way you break out of the Foreach loop. In the following example, the Break statement exits a While statement when a DivideByZeroException exception is trapped using the Trap statement.

```
$i = 3
while ($true)
{
   trap [DivideByZeroException]
   {
      Write-Host "divide by zero trapped"
      break
   }
   1 / $i--
}
```

A Break statement can include a label. If you use the Break keyword with a label, Windows PowerShell exits the labeled loop instead of exiting the current loop. The syntax for a label is as follows (this example shows a label in a While loop):

```
:myLabel while (<condition>) { <statement list>}
```

The label is a colon followed by a name that you assign. The label must be the first token in a statement, and it must be followed by the looping keyword, such as While.

In Windows PowerShell, only loop keywords, such as Foreach, For, and While can have a label.

Break moves execution out of the labeled loop. In embedded loops, this has a different result than the Break keyword has when it is used by itself. This schematic example has a While statement with a For statement:

```
:myLabel while (<condition 1>)
{
   for ($item in $items)
   {
      if (<condition 2>) { break myLabel }
      $item = $x   # A statement inside the For-loop
   }
}
$a = $c  # A statement after the labeled While-loop
```

If condition 2 evaluates to True, the execution of the script skips down to the statement after the labeled loop. In the example, execution starts again with the statement "$a = $c".

You can nest many labeled loops, as shown in the following schematic example.

```
:red while (<condition1>)
{
   :yellow while (<condition2>)
   {
      while (<condition3>)
      {
         if ($a) {break}
         if ($b) {break red}
         if ($c) {break yellow}
      }
      # After innermost loop
   }
      # After "yellow" loop
}
      # After "red" loop
```

If the $b variable evaluates to True, execution of the script resumes after the loop that is labeled "red". If the $c variable evaluates to True, execution of the script control resumes after the loop that is labeled "yellow".

If the $a variable evaluates to True, execution resumes after the innermost loop. No label is needed.

Windows PowerShell does not limit how far labels can resume execution. The

label can even pass control across script and function call boundaries.

The Break keyword is used to leave the Switch construct. For example, the following Switch statement uses Break statements to test for the most specific condition:

```
$var = "word2"
switch -regex ($var)
{
   "word2"
   {
      Write-Host "Exact" $_
      break
   }

   "word.*"
   {
      Write-Host "Match on the prefix" $_
      break
   }

   "w.*"
   {
      Write-Host "Match on at least the first letter" $_
      break
   }

   default
   {
      Write-Host "No match" $_
      break
   }
}
```

In this example, the $var variable is created and initialized to a string value of "word2". The Switch statement uses the Regex class to match the variable value first with the term "word2". (The Regex class is a regular expression Microsoft .NET Framework class.) Because the variable value and the first test in the Switch statement match, the first code block in the Switch statement runs.

When Windows PowerShell reaches the first Break statement, the Switch statement exits. If the four Break statements are removed from the example, all four conditions are met. This example uses the break statement to

display results when the most specific condition is met.

SEE ALSO
about_Comparison_Operators
about_Continue
about_For
about_Foreach
about_Switch
about_Throw
about_Trap
about_Try_Catch_Finally
about_While

```
Name                                Category  Module              Synopsis
----                                --------  ------              --------
about_Checkpoint-Workflow           HelpFile                      Describes the Checkpoint-Workflow activity,
which
about_Checkpoint-Workflow           HelpFile                      Describes the Checkpoint-Workflow activity,
which
```

about_CimSession

SHORT DESCRIPTION
Describes a CimSession object and the difference between CIM sessions and Windows PowerShell sessions.

LONG DESCRIPTION
A Common Information Model (CIM) session is a client-side object that represents a connection to a local computer or a remote computer. You can use CIM sessions as an alternative to Windows PowerShell sessions (PSSessions). Both approaches have advantages.

You can use the New-CimSession cmdlet to create a CIM session that contains information about a connection, such as computer name, the protocol used for the connection, session ID, and instance ID.

After you create a CimSession object that specifies information required to establish a connection, Windows PowerShell does not establish the connection immediately. When a cmdlet uses the CIM session, Windows PowerShell connects to the specified computer, and then, when the cmdlet finishes, Windows PowerShell terminates the connection.

If you create a PSSession instead of using a CIM session, Windows PowerShell validates connection settings, and then establishes and maintains the connection. If you use CIM sessions, Windows PowerShell does not open a network connection until needed. For more information about Windows PowerShell sessions, see about_PSSessions.

When to Use a CIM Session

Only cmdlets that work with a Windows Management Infrastructure (WMI) provider accept CIM sessions. For other cmdlets, use PSSessions.

When you use a CIM session, Windows PowerShell runs the cmdlet on the local client. It connects to the WMI provider by using the CIM session. The target computer does not require Windows PowerShell, or even any version of the Windows operating system.

In contrast, a cmdlet run by using a PSSession runs on the target computer. It requires Windows PowerShell on the target system. Furthermore, the cmdlet sends data back to the local computer. Windows PowerShell manages the data sent over the connection, and keeps the size within the limits set by Windows Remote Management (WinRM). CIM sessions do not impose the WinRM limits.

PSSessions only work with WinRM. CimSessions can use DCOM.

CIM-based Cmdlet Definition XML (CDXML) cmdlets can be written to use any WMI Provider. All WMI providers use CimSession objects.

SEE ALSO

New-CimSession

about_PSSessions

about_Classes

SHORT DESCRIPTION

Describes how you can use classes to develop in Windows PowerShell

LONG DESCRIPTION

Starting in Windows PowerShell 5.0, Windows PowerShell adds language for defining classes and other user-defined types, by using formal syntax and semantics that are similar to other object-oriented programming languages. The goal is to enable developers and IT professionals to embrace Windows PowerShell for a wider range of use cases, simplify development of Windows PowerShell artifacts--such as Windows PowerShell Desired State Configuration (DSC) resources--and accelerate coverage of management surfaces.

SUPPORTED SCENARIOS

-- Define DSC resources and their associated types by using the Windows PowerShell language.
-- Define custom types in Windows PowerShell by using familiar object-oriented programming constructs, such as classes, properties, methods, inheritance, etc.
-- Debug types by using the Windows PowerShell language.
-- Generate and handle exceptions by using formal mechanisms, and at the right level.

DEFINE DSC RESOURCES WITH CLASSES

Apart from syntax changes, the major differences between a class-defined DSC resource and a cmdlet DSC resource provider are the following.

-- A MOF file is not required.
-- A DSCResource subfolder in the module folder is not required.
-- A Windows PowerShell module file can contain multiple DSC resource classes.

The following is an example of a class-defined DSC resource provider; this is saved as a module, MyDSCResource.psm1. Note that you must always include a key property in a class-defined DSC resource provider.

```
enum Ensure
{
   Absent
   Present
```

```
}

<#
  This resource manages the file in a specific path.
  [DscResource()] indicates the class is a DSC resource
#>

 [DscResource()]
class FileResource
{
   <#
     This property is the fully qualified path to the file that is
     expected to be present or absent.

     The [DscProperty(Key)] attribute indicates the property is a
     key and its value uniquely identifies a resource instance.
     Defining this attribute also means the property is required
     and DSC will ensure a value is set before calling the resource.

     A DSC resource must define at least one key property.
   #>
   [DscProperty(Key)]
   [string]$Path

   <#
      This property indicates if the settings should be present or absent
      on the system. For present, the resource ensures the file pointed
      to by $Path exists. For absent, it ensures the file point to by
      $Path does not exist.

      The [DscProperty(Mandatory)] attribute indicates the property is
      required and DSC will guarantee it is set.

      If Mandatory is not specified or if it is defined as
      Mandatory=$false, the value is not guaranteed to be set when DSC
      calls the resource.  This is appropriate for optional properties.
   #>
   [DscProperty(Mandatory)]
   [Ensure] $Ensure

   <#
     This property defines the fully qualified path to a file that will
     be placed on the system if $Ensure = Present and $Path does not
     exist.

     NOTE: This property is required because [DscProperty(Mandatory)] is
      set.
   #>
```

```
[DscProperty(Mandatory)]
[string] $SourcePath

<#
  This property reports the file's create timestamp.

  [DscProperty(NotConfigurable)] attribute indicates the property is
  not configurable in DSC configuration.  Properties marked this way
  are populated by the Get() method to report additional details
  about the resource when it is present.

#>
[DscProperty(NotConfigurable)]
[Nullable[datetime]] $CreationTime

<#
   This method is equivalent of the Set-TargetResource script function.
   It sets the resource to the desired state.
#>
[void] Set()
{
   $fileExists = $this.TestFilePath($this.Path)
   if($this.ensure -eq [Ensure]::Present)
   {
      if(-not $fileExists)
      {
         $this.CopyFile()
      }
   }
   else
   {
      if($fileExists)
      {
         Write-Verbose -Message "Deleting the file $($this.Path)"
         Remove-Item -LiteralPath $this.Path -Force
      }
   }
}

<#

   This method is equivalent of the Test-TargetResource script function.
   It should return True or False, showing whether the resource
   is in a desired state.
#>
[bool] Test()
{
   $present = $this.TestFilePath($this.Path)
```

```
    if($this.Ensure -eq [Ensure]::Present)
    {
       return $present
    }
    else
{
       return -not $present
    }
 }

 <#
    This method is equivalent of the Get-TargetResource script function.
    The implementation should use the keys to find appropriate resources.
    This method returns an instance of this class with the updated key
     properties.
 #>
 [FileResource] Get()
 {
    $present = $this.TestFilePath($this.Path)

    if ($present)
    {
       $file = Get-ChildItem -LiteralPath $this.Path
       $this.CreationTime = $file.CreationTime
       $this.Ensure = [Ensure]::Present
    }
    else
    {
       $this.CreationTime = $null
       $this.Ensure = [Ensure]::Absent
    }
    return $this
 }

 <#
    Helper method to check if the file exists and it is correct file
 #>
 [bool] TestFilePath([string] $location)
 {
    $present = $true

    $item = Get-ChildItem -LiteralPath $location -ea Ignore
    if ($item -eq $null)
    {
       $present = $false
    }
    elseif( $item.PSProvider.Name -ne "FileSystem")
```

```
        {
            throw "Path $($location) is not a file path."
        }
        elseif($item.PSIsContainer)
        {
            throw "Path $($location) is a directory path."
        }
        return $present
    }

    <#
        Helper method to copy file from source to path
    #>
    [void] CopyFile()
    {
        if(-not $this.TestFilePath($this.SourcePath))
        {
            throw "SourcePath $($this.SourcePath) is not found."
        }

        [System.IO.FileInfo] $destFileInfo = new-object System.IO.FileInfo($this.Path)
        if (-not $destFileInfo.Directory.Exists)
        {
            Write-Verbose -Message "Creating directory $($destFileInfo.Directory.FullName)"

            #use CreateDirectory instead of New-Item to avoid code
            # to handle the non-terminating error
            [System.IO.Directory]::CreateDirectory($destFileInfo.Directory.FullName)
        }

        if(Test-Path -LiteralPath $this.Path -PathType Container)
        {
            throw "Path $($this.Path) is a directory path"
        }

        Write-Verbose -Message "Copying $($this.SourcePath) to $($this.Path)"

        #DSC engine catches and reports any error that occurs
        Copy-Item -LiteralPath $this.SourcePath -Destination $this.Path -Force
    }
}
# This module defines a class for a DSC "FileResource" provider.


enum Ensure
{
    Absent
    Present
```

```
}


<# This resource manages the file in a specific path.
 [DscResource()] indicates the class is a DSC resource
#>

 [DscResource()]
class FileResource{

   <# This is a key property
     [DscResourceKey()] also means the property is required.
     It is guaranteed to be set, other properties may not
      be set if the configuration did not specify values.
   #>
   [DscResourceKey()]
   [string]$Path

   <#
     [DscResourceMandatory()] means the property is required.
     It is guaranteed to be set, other properties may not be set
      if the configuration did not specify values.
   #>
   [DscResourceMandatory()]
   [Ensure] $Ensure

   <#
     [DscResourceMandatory()] means the property is required.
   #>
   [DscResourceMandatory()]
   [string] $SourcePath

   [DscResource

   <#
      This method replaces the Set-TargetResource DSC script function.
      It sets the resource to the desired state.
   #>
   [void] Set()
   {
      $fileExists = Test-Path -path $this.Path -PathType Leaf
      if($this.ensure -eq [Ensure]::Present)
      {
         if(-not $fileExists)
         {
            $this.CopyFile()
         }
      }
```

```
    else
    {
        if($fileExists)
        {
            Write-Verbose -Message "Deleting the file $this.Path"
            Remove-Item -LiteralPath $this.Path
        }
    }
}


<#

    This method replaces the Test-TargetResource function.
    It should return True or False, showing whether the resource
     is in a desired state.
#>

[bool] Test()
{
    if(Test-Path -path $this.Path -PathType Container)
    {
        throw "Path '$this.Path' is a directory path."
    }

    $fileExists = Test-Path -path $this.Path -PathType Leaf

    if($this.ensure -eq [Ensure]::Present)
    {
        return $fileExists
    }

    return (-not $fileExists)
}

<#
    This method replaces the Get-TargetResource function.
    The implementation should use the keys to find appropriate resources.
    This method returns an instance of this class with the updated key properties.
#>

[FileResource] Get()
{
    $file = Get-item $this.Path
    return $this
}

<#
```

```
        Helper method to copy file from source to path.
        Because this resource provider run under system,
        Only the Administrators and system have full
        access to the new created directory and file
    #>
    CopyFile()
    {
        if(Test-Path -path $this.SourcePath -PathType Container)
        {
            throw "SourcePath '$this.SourcePath' is a directory path"
        }

        if( -not (Test-Path -path $this.SourcePath -PathType Leaf))
        {
            throw "SourcePath '$this.SourcePath' is not found."
        }

        [System.IO.FileInfo] $destFileInfo = new-object System.IO.FileInfo($this.Path)
        if (-not $destFileInfo.Directory.Exists)
        {
            Write-Verbose -Message "Creating directory $($destFileInfo.Directory.FullName)"

            #use CreateDirectory instead of New-Item to avoid lines
            # to handle the non-terminating error
            [System.IO.Directory]::CreateDirectory($destFileInfo.Directory.FullName)
        }

        if(Test-Path -path $this.Path -PathType Container)
        {
            throw "Path '$this.Path' is a directory path"
        }

        Write-Verbose -Message "Copying $this.SourcePath to $this.Path"

        #DSC engine catches and reports any error that occurs
        Copy-Item -Path $this.SourcePath -Destination $this.Path -Force
    }
}
```

After creating the class-defined DSC resource provider, and saving it as a module, create a module manifest for the module. To make a class-based resource available to the DSC engine, you must include a DscResourcesToExport statement in the manifest file that instructs the module to export the resource. In this example, the following module manifest is saved as MyDscResource.psd1.

```
@{

# Script module or binary module file associated with this manifest.
```

```
RootModule = 'MyDscResource.psm1'

DscResourcesToExport = 'FileResource'

# Version number of this module.
ModuleVersion = '1.0'

# ID used to uniquely identify this module
GUID = '81624038-5e71-40f8-8905-b1a87afe22d7'

# Author of this module
Author = 'Microsoft Corporation'

# Company or vendor of this module
CompanyName = 'Microsoft Corporation'

# Copyright statement for this module
Copyright = '(c) 2014 Microsoft. All rights reserved.'

# Description of the functionality provided by this module
# Description = ''

# Minimum version of the Windows PowerShell engine required by this module
PowerShellVersion = '5.0'

# Name of the Windows PowerShell host required by this module
# PowerShellHostName = ''

}
```

Deploy the new DSC resource provider by creating a MyDscResource folder for it in $pshome\Modules or $env:SystemDrive\Program Files\WindowsPowerShell\Modules. You do not need to create a DSCResource subfolder. Copy the module and module manifest files (MyDscResource.psm1 and MyDscResource.psd1) to the MyDscResource folder.

From this point, you create and run a configuration script as you would with any DSC resource. The following is a configuration that references the MyDSCResource module. Save this as a script, MyResource.ps1.

INHERITANCE IN WINDOWS POWERSHELL CLASSES

Declare base classes for Windows PowerShell classes

You can declare a Windows PowerShell class as a base type for another Windows PowerShell class, as shown in the following example, in which "fruit" is a base type for "apple".

```
class fruit
{
   [int]sold() {return 100500}
}

class apple : fruit {}
 [apple]::new().sold() # return 100500
```

Declare implemented interfaces for Windows PowerShell classes

You can declare implemented interfaces after base types, or immediately after a colon (:) if there is no base type specified. Separate all type names by using commas. This is similar to C# syntax.

```
class MyComparable : system.IComparable
{
   [int] CompareTo([object] $obj)
   {
      return 0;
   }
}

class MyComparableTest : test, system.IComparable
{
   [int] CompareTo([object] $obj)
   {
      return 0;
   }
}
```

Call base class constructors

To call a base class constructor from a subclass, add the "base" keyword, as shown in the following example.

```
class A {
   [int]$a
   A([int]$a)
   {
      $this.a = $a
   }
}

class B : A
{
   B() : base(103) {}
}

 [B]::new().a # return 103
```

If a base class has a default constructor (that is, no parameters), you can omit an explicit constructor call, as shown.

```
class C : B
{
   C([int]$c) {}
}
```

Call base class methods

You can override existing methods in subclasses. To do this, declare methods by using the same name and signature.

```
class baseClass
{
   [int]days() {return 100500}
}
class childClass1 : baseClass
{
   [int]days () {return 200600}
}

 [childClass1]::new().days() # return 200600
```

To call base class methods from overridden implementations, cast to the base class ([baseclass]$this) on invocation.

```
class childClass2 : baseClass
{
   [int]days()
   {
      return 3 * ([baseClass]$this).days()
   }
}

 [childClass2]::new().days() # return 301500
```

All Windows PowerShell methods are virtual. You can hide non-virtual .NET methods in a subclass by using the same syntax as you do for an override: declare methods with same name and signature.

```
class MyIntList : system.collections.generic.list[int]
{
   # Add is final in system.collections.generic.list
   [void] Add([int]$arg)
   {
      ([system.collections.generic.list[int]]$this).Add($arg * 2)
   }
}
```

```
$list = [MyIntList]::new()
$list.Add(100)
$list[0] # return 200
```

Current limitations with inheritance

The following are known issues with class inheritance:

-- At this time, there is no syntax to declare interfaces in Windows PowerShell.

CONFIGURATION TEST

After saving the class and manifest files in the folder structure as described earlier, you can create a configuration that uses the new resource. For information about how to run a DSC configuration, see Get Started with Windows PowerShell Desired State Configuration (http://technet.microsoft.com/library/dn249918.aspx). The following configuration will check to see whether the file at c:\test\test.txt exists, and, if not, copies the file from c:\test.txt (you should create c:\test.txt before you run the configuration).

```
Configuration Test
{
   Import-DSCResource -module MyDscResource
   FileResource file
   {
      Path = "C:\test\test.txt"
      SourcePath = "C:\test.txt"
      Ensure = "Present"
   }
}
Test
Start-DscConfiguration -Wait -Force Test
```

Run this as you would any DSC configuration script. To start the configuration, in an elevated Windows PowerShell console, run the following.

```
PS C:\test> .\MyResource.ps1
```

DEFINING CUSTOM TYPES IN WINDOWS POWERSHELL

Windows PowerShell 5.0 introduces the following language elements.

Class keyword

Defines a new class. This is a true .NET Framework type. Class members are public.

```
class MyClass
{
}
```

Enum keyword and enumerations

Support for the enum keyword has been added; this is a breaking change. The enum delimiter is currently a newline. A workaround for those who are already using enum is to insert an ampersand (&) before the word. Current limitations: you cannot define an enumerator in terms of itself, but you can initialize enum in terms of another enum, as shown in the following example. The base type cannot currently be specified; it is always [int].

```
enum Color2
{
   Yellow = [Color]::Blue
}
```

An enumerator value must be a parse time constant; you cannot set it to the result of an invoked command.

```
enum MyEnum
{
   Enum1
   Enum2
   Enum3 = 42
   Enum4 = [int]::MaxValue
}
```

Enums support arithmetic operations, as shown in the following example.

```
enum SomeEnum { Max = 42 }
enum OtherEnum { Max = [SomeEnum]::Max + 1 }
```

Hidden keyword

The Hidden keyword, new in Windows PowerShell 5.0, hides class members from default Get-Member results. Specify the hidden property as shown in the following line:

```
hidden [type] $classmember = <value>
```

Hidden members are not displayed by using tab completion or IntelliSense, unless the completion occurs in the class that defines the hidden member.

A new attribute, System.Management.Automation.HiddenAttribute, has been added, so that C# code can have the same semantics within Windows PowerShell.

For more information about Hidden, see about_Hidden.

Import-DscResource

Import-DscResource is now a true dynamic keyword. Windows PowerShell parses the specified module's root module, searching for classes that contain the DscResource attribute.

Properties

A new field, ImplementingAssembly, has been added to ModuleInfo. It is set to the dynamic assembly created for a script module if the script defines classes, or the loaded assembly for binary modules. It is not

set when ModuleType = Manifest.

Reflection on the ImplementingAssembly field discovers resources in a module. This means you can discover resources written in either Windows PowerShell or other managed languages.

Fields with initializers:

```
[int] $i = 5
```

Static is supported; it works like an attribute, as do the type constraints, so it can be specified in any order.

```
static [int] $count = 0
```

A type is optional.

```
$s = "hello"
```

All members are public. Properties require either a newline or semicolon. If no object type is specified, the property type is “object.”

Constructors and instantiation

Windows PowerShell classes can have constructors; they have the same name as their class. Constructors can be overloaded. Static constructors are supported. Properties with initialization expressions are initialized before running any code in a constructor. Static properties are initialized before the body of a static constructor, and instance properties are initialized before the body of the non-static constructor. Currently, there is no syntax for calling a constructor from another constructor (like the C# syntax ": this()"). The workaround is to define a common Init method.

The following are ways of instantiating classes.

Instantiating by using the default constructor. Note that New-Object is not supported in this release.

```
$a = [MyClass]::new()
```

Calling a constructor with a parameter

```
$b = [MyClass]::new(42)
```

Passing an array to a constructor with multiple parameters

```
$c = [MyClass]::new(@(42,43,44), "Hello")
```

For this release, the type name is only visible lexically, meaning it is not visible outside of the module or script that defines the class. Functions can return instances of a class defined in Windows PowerShell, and instances work well outside of the module or script.

Get-Member -Static lists constructors, so you can view overloads like any other method. The performance of this syntax is also considerably faster than New-Object.

The pseudo-static method named new works with .Net types, as shown in

the following example.

```
[hashtable]::new()
```

You can now see constructor overloads with Get-Member, or as shown in this example:

```
PS> [hashtable]::new
OverloadDefinitions
-------------------
hashtable new()
hashtable new(int capacity)
hashtable new(int capacity, float loadFactor)
```

Methods

A Windows PowerShell class method is implemented as a ScriptBlock that has only an end block. All methods are public. The following shows an example of defining a method named DoSomething.

```
class MyClass
{
   DoSomething($x)
   {
      $this._doSomething($x)      # method syntax
   }
   private _doSomething($a) {}
}
```

Method invocation:

```
$b = [MyClass]::new()
$b.DoSomething(42)
```

Overloaded methods--that is, those that are named the same as an existing method, but differentiated by their specified values--are also supported.

Invocation

See "Method invocation" in this list.

Attributes

Three new attributes, DscResource, DscResourceKey, and DscResourceMandatory, have been added.

Return types

Return type is a contract; the return value is converted to the expected type. If no return type is specified, the return type is void. There is no streaming of objects; objects cannot be written to the pipeline either intentionally or by accident.

Lexical scoping of variables

The following shows an example of how lexical scoping works in this release.

```
$d = 42  # Script scope

function bar
{
   $d = 0  # Function scope
   [MyClass]::DoSomething()
}

class MyClass
{
   static [object] DoSomething()
   {
      return $d  # error, not found dynamically
      return $script:d # no error

      $d = $script:d
      return $d # no error, found lexically
   }
}

$v = bar
$v -eq $d # true
```

EXAMPLE

The following example creates several new, custom classes to implement an HTML dynamic stylesheet language (DSL). Then, the example adds helper functions to create specific element types as part of the element class, such as heading styles and tables, because types cannot be used outside the scope of a module.

```
# Classes that define the structure of the document
#
class Html
{
   [string] $docType
   [HtmlHead] $Head
   [Element[]] $Body

   [string] Render()
   {
      $text = "<html>`n<head>`n"
      $text += $Head
      $text += "`n</head>`n<body>`n"
      $text += $Body -join "`n" # Render all of the body elements
      $text += "</body>`n</html>"
```

```
        return $text
    }
    [string] ToString() { return $this.Render() }
}

class HtmlHead
{
    $Title
    $Base
    $Link
    $Style
    $Meta
    $Script
    [string] Render() { return "<title>$Title</title>" }
    [string] ToString() { return $this.Render() }
}

class Element
{
    [string] $Tag
    [string] $Text
    [hashtable] $Attributes
    [string] Render() {
        $attributesText= ""
        if ($Attributes)
        {
            foreach ($attr in $Attributes.Keys)
            {
                $attributesText = " $attr=`"$($Attributes[$attr])`""
            }
        }

        return "<${tag}${attributesText}>$text</$tag>`n"
    }
    [string] ToString() { return $this.Render() }
}

#
# Helper functions for creating specific element types on top of the classes.
# These are required because types aren't visible outside of the module.
#
function H1 { [Element] @{ Tag = "H1" ; Text = $args.foreach{$_} -join " " }}
function H2 { [Element] @{ Tag = "H2" ; Text = $args.foreach{$_} -join " " }}
function H3 { [Element] @{ Tag = "H3" ; Text = $args.foreach{$_} -join " " }}
function P  { [Element] @{ Tag = "P"  ; Text = $args.foreach{$_} -join " " }}
function B  { [Element] @{ Tag = "B"  ; Text = $args.foreach{$_} -join " " }}
function I  { [Element] @{ Tag = "I"  ; Text = $args.foreach{$_} -join " " }}
function HREF
```

```
{
   param (
      $Name,
      $Link
   )

   return [Element] @{
      Tag = "A"
      Attributes = @{ HREF = $link }
      Text = $name
   }
}
function Table
{
   param (
      [Parameter(Mandatory)]
      [object[]]
         $Data,
      [Parameter()]
      [string[]]
         $Properties = "*",
      [Parameter()]
      [hashtable]
         $Attributes = @{ border=2; cellpadding=2; cellspacing=2 }
   )

   $bodyText = ""
   # Add the header tags
   $bodyText +=  $Properties.foreach{TH $_}
   # Add the rows
   $bodyText += foreach ($row in $Data)
            {
                  TR (-join $Properties.Foreach{ TD ($row.$_) } )
            }


   $table = [Element] @{
            Tag = "Table"
            Attributes = $Attributes
            Text = $bodyText
          }
   $table
}
function TH  { ([Element] @{ Tag = "TH"  ; Text = $args.foreach{$_} -join " " }) }
function TR  { ([Element] @{ Tag = "TR"  ; Text = $args.foreach{$_} -join " " }) }
function TD  { ([Element] @{ Tag = "TD"  ; Text = $args.foreach{$_} -join " " }) }

function Style
```

```
    {
      return  [Element]  @{
        Tag = "style"
        Text = "$args"
      }
    }

    # Takes a hash table, casts it to and HTML document
    # and then returns the resulting type.
    #
    function Html ([HTML] $doc) { return $doc }
```

SEE ALSO

about_DesiredStateConfiguration
about_Language_Keywords
about_Methods
Writing a custom DSC resource with PowerShell classes
(http://technet.microsoft.com/library/dn948461.aspx)

about_Command_Precedence

SHORT DESCRIPTION

Describes how Windows PowerShell determines which command to run.

LONG DESCRIPTION

This topic explains how Windows PowerShell determines which command to run, especially when a session contains more than one command with the same name. It also explains how to run commands that do not run by default, and it explains how to avoid command-name conflicts in your session.

COMMAND PRECEDENCE

When a session includes commands that have the same name, Windows PowerShell uses the following rules to decide which command to run.

These rules become very important when you add commands to your session from modules, snap-ins, and other sessions.

-- If you specify the path to a command, Windows PowerShell runs the command at the location specified by the path.

For example, the following command runs the FindDocs.ps1 script in the C:\TechDocs directory:

```
C:\TechDocs\FindDocs.ps1
```

As a security feature, Windows PowerShell does not run executable (native) commands, including Windows PowerShell scripts, unless the command is located in a path that is listed in the Path environment variable ($env:path) or unless you specify the path to the script file.

To run a script that is in the current directory, specify the full path, or type a dot (.) to represent the current directory.

For example, to run the FindDocs.ps1 file in the current directory, type:

```
.\FindDocs.ps1
```

-- If you do not specify a path, Windows PowerShell uses the following precedence order when it runs commands:

1. Alias
2. Function
3. Cmdlet
4. Native Windows commands

Therefore, if you type "help", Windows PowerShell first looks for an alias named "help", then a function named "Help", and finally a cmdlet named "Help". It runs the first "help" item that it finds.

For example, if you have a Get-Map function in the session and you import a cmdlet named Get-Map. By default, when you type "Get-Map", Windows PowerShell runs the Get-Map function.

-- When the session contains items of the same type that have the same name, such as two cmdlets with the same name, Windows PowerShell runs the item that was added to the session most recently.

For example, if you have a cmdlet named Get-Date and you import another cmdlet named Get-Date, by default, Windows PowerShell runs the most-recently imported cmdlet when you type "Get-Date".

HIDDEN and REPLACED ITEMS

As a result of these rules, items can be replaced or hidden by items with

the same name.

-- Items are "hidden" or "shadowed" if you can still access the original item, such as by qualifying the item name with a module or snap-in name.

For example, if you import a function that has the same name as a cmdlet in the session, the cmdlet is hidden (but not replaced) because it was imported from a snap-in or module.

-- Items are "replaced" or "overwritten" if you can no longer access the original item.

For example, if you import a variable that has the same name as a a variable in the session, the original variable is replaced and is no longer accessible. You cannot qualify a variable with a module name.

Also, if you type a function at the command line and then import a function with the same name, the original function is replaced and is no longer accessible.

FINDING HIDDEN COMMANDS

The All parameter of the Get-Command cmdlet gets all commands with the specified name, even if they are hidden or replaced. Beginning in Windows PowerShell 3.0, by default, Get-Command gets only the commands that run when you type the command name.

In the following examples, the session includes a Get-Date function and a Get-Date cmdlet.

The following command gets the Get-Date command that runs when you type "Get-Date".

```
PS C:\> Get-Command Get-Date

CommandType     Name                                ModuleName
-----------     ----                                ----------
Function        get-date
```

The following command uses the All parameter to get all Get-Date commands.

```
PS C:\> Get-Command Get-Date -All

CommandType     Name                                ModuleName
-----------     ----                                ----------
Function        get-date
```

Cmdlet Get-Date Microsoft.PowerShell.Utility

RUNNING HIDDEN COMMANDS

You can run particular commands by specifying item properties that distinguish the command from other commands that might have the same name.

You can use this method to run any command, but it is especially useful for running hidden commands.

Use this method as a best practice when writing scripts that you intend to distribute because you cannot predict which commands might be present in the session in which the script runs.

QUALIFIED NAMES

You can run commands that have been imported from a Windows PowerShell snap-in or module or from another session by qualifying the command name with the name of the module or snap-in in which it originated.

You can qualify commands, but you cannot qualify variables or aliases.

For example, if the Get-Date cmdlet from the Microsoft.PowerShell.Utility snap-in is hidden by an alias, function, or cmdlet with the same name, you can run it by using the snap-in-qualified name of the cmdlet:

```
Microsoft.PowerShell.Utility\Get-Date
```

To run a New-Map command that was added by the MapFunctions module, use its module-qualified name:

```
MapFunctions\New-Map
```

To find the snap-in or module from which a command was imported, use the ModuleName property of commands.

```
(Get-Command <command-name>).ModuleName
```

For example, to find the source of the Get-Date cmdlet, type:

```
PS C:\>(Get-Command Get-Date).ModuleName
Microsoft.PowerShell.Utility
```

CALL OPERATOR

You can also use the Call operator (&) to run any command that you can get by using a Get-ChildItem (the alias is "dir"), Get-Command, or Get-Module command.

To run a command, enclose the Get-Command command in parentheses, and use the Call operator (&) to run the command.

```
&(Get-Command ...)
```

- or -

```
&(dir ... )
```

For example, if you have a function named Map that is hidden by an alias named Map, use the following command to run the function.

```
&(Get-Command -Name Map -Type function)
```

- or -

```
&(dir function:\map)
```

You can also save your hidden command in a variable to make it easier to run.

For example, the following command saves the Map function in the $myMap variable and then uses the Call operator to run it.

```
$myMap = (Get-Command -Name map -Type function)

&($myMap)
```

If a command originated in a module, you can use the following format to run it.

```
& <PSModuleInfo-object> <command>
```

For example, to run the Add-File cmdlet in the FileCommands module, use the following command sequence.

```
$FileCommands = get-module -name FileCommands

& $FileCommands Add-File
```

REPLACED ITEMS

Items that have not been imported from a module or snap-in, such as functions, variables, and aliases that you create in your session or that you add by using a profile can be replaced by commands that have the same name. If they are replaced, you cannot access them.

Variables and aliases are always replaced even if they have been imported from a module or snap-in because you cannot use a call operator or a qualified name to run them.

For example, if you type a Get-Map function in your session, and you import a function called Get-Map, the original function is replaced. You cannot retrieve it in the current session.

AVOIDING NAME CONFLICTS

The best way to manage command name conflicts is to prevent them. When you name your commands, use a name that is very specific or is likely to be unique. For example, add your initials or company name acronym to the nouns in your commands.

Also, when you import commands into your session from a Windows PowerShell module or from another session, use the Prefix parameter of the Import-Module or Import-PSSession cmdlet to add a prefix to the nouns in the names of commands.

For example, the following command avoids any conflict with the Get-Date and Set-Date cmdlets that come with Windows PowerShell when you import the DateFunctions module.

```
Import-Module -Name DateFunctions -Prefix ZZ
```

For more information, see Import-Module and Import-PSSession.

SEE ALSO

about_Path_Syntax
about_Aliases
about_Functions
Alias (provider)
Function (provider)
Get-Command
Import-Module
Import-PSSession

about_Command_Syntax

SHORT DESCRIPTION

Describes the syntax diagrams that are used in Windows PowerShell.

LONG DESCRIPTION

The Get-Help and Get-Command cmdlets display syntax diagrams to help you construct commands correctly. This topic explains how to interpret the syntax diagrams.

Syntax Diagrams

Each paragraph in a command syntax diagram represents a valid form of the command.

To construct a command, follow the syntax diagram from left to right. Select from among the optional parameters and provide values for the placeholders.

Windows PowerShell uses the following notation for syntax diagrams.

```
<command-name> -<Required Parameter Name> <Required Parameter Value>
        [-<Optional Parameter Name> <Optional Parameter Value>]
        [-<Optional Switch Parameters>]
        [-<Optional Parameter Name>] <Required Parameter Value>
```

The following is the syntax for the New-Alias cmdlet.

```
New-Alias [-Name] <string> [-Value] <string> [-Description <string>]
  [-Force] [-Option {None | ReadOnly | Constant | Private | AllScope}]
  [-PassThru] [-Scope <string>] [-Confirm] [-WhatIf] [<CommonParameters>]
```

The syntax is capitalized for readability, but Windows PowerShell is case-insensitive.

The syntax diagram has the following elements.

Command name

Commands always begin with a command name, such as New-Alias. Type the command name or its alias, such a "gcm" for Get-Command.

Parameters

The parameters of a command are options that determine what the command does. Some parameters take a "value," which is user input to the command.

For example, the Get-Help command has a Name parameter that lets you specify the name of the topic for which help is displayed. The topic name is the value of the Name parameter.

In a Windows PowerShell command, parameter names always begin with a hyphen. The hyphen tells Windows PowerShell that the item in the command is a parameter name.

For example, to use the Name parameter of New-Alias, you type the following:

    -Name

Parameters can be mandatory or optional. In a syntax diagram, optional items are enclosed in brackets ([ ]).

For more information about parameters, see about_Parameters.

Parameter Values

A parameter value is the input that the parameter takes. Because Windows PowerShell is based on the Microsoft .NET Framework, parameter values are represented in the syntax diagram by their .NET type.

For example, the Name parameter of Get-Help takes a String value, which is a text string, such as a single word or multiple words enclosed in quotation marks.

    [-Name] <string>

The .NET type of a parameter value is enclosed in angle brackets (< >) to indicate that it is placeholder for a value and not a literal that you type in a command.

To use the parameter, replace the .NET type placeholder with an object that has the specified .NET type.

For example, to use the Name parameter, type "-Name" followed by a string, such as the following:

    -Name MyAlias

Parameters with no values

Some parameters do not accept input, so they do not have a parameter value. Parameters without values are called "switch parameters" because they work like on/off switches. You include them (on) or you omit them (off) from a command.

To use a switch parameter, just type the parameter name, preceded by a hyphen.

For example, to use the WhatIf parameter of the New-Alias cmdlet, type the following:

```
-WhatIf
```

Parameter Sets

The parameters of a command are listed in parameter sets. Parameter sets look like the paragraphs of a syntax diagram.

The New-Alias cmdlet has one parameter set, but many cmdlets have multiple parameter sets. Some of the cmdlet parameters are unique to a parameter set, and others appear in multiple parameter sets.

Each parameter set represents the format of a valid command. A parameter set includes only parameters that can be used together in a command. If parameters cannot be used in the same command, they appear in separate parameter sets.

For example, the Get-Random cmdlet has the following parameter sets:

```
Get-Random [[-Maximum] <Object>] [-Minimum <Object>] [-SetSeed <int>]
        [<CommonParameters>]

Get-Random [-InputObject] <Object[]> [-Count <int>] [-SetSeed <int>]
        [<CommonParameters>]
```

The first parameter set, which returns a random number, has the Minimum and Maximum parameters. The second parameter set, which returns a randomly selected object from a set of objects, includes the InputObject and Count parameters. Both parameter sets have the Set-Seed parameter and the common parameters.

These parameter sets indicate that you can use the InputObject and Count parameters in the same command, but you cannot use the Maximum and Count parameters in the same command.

You indicate which parameter set you want to use by using the parameters in that parameter set.

However, every cmdlet also has a default parameter set. The default parameter

set is used when you do not specify parameters that are unique to a parameter set. For example, if you use Get-Random without parameters, Windows PowerShell assumes that you are using the Number parameter set and it returns a random number.

In each parameter set, the parameters appear in position order. The order of parameters in a command matters only when you omit the optional parameter names. When parameter names are omitted, Windows PowerShell assigns values to parameters by position and type. For more information about parameter position, see about_Parameters.

Symbols in Syntax Diagrams

The syntax diagram lists the command name, the command parameters, and the parameter values. It also uses symbols to show how to construct a valid command.

The syntax diagrams use the following symbols:

-- A hyphen (-) indicates a parameter name. In a command, type the hyphen immediately before the parameter name with no intervening spaces, as shown in the syntax diagram.

For example, to use the Name parameter of New-Alias, type:

```
-Name
```

-- Angle brackets (<>) indicate placeholder text. You do not type the angle brackets or the placeholder text in a command. Instead, you replace it with the item that it describes.

Angle brackets are used to identify the .NET type of the value that a parameter takes. For example, to use the Name parameter of the New-Alias cmdlet, you replace the <string> with a string, which is a single word or a group of words that are enclosed in quotation marks.

-- Brackets ([ ]) indicate optional items. A parameter and its value can be optional, or the name of a required parameter can be optional.

For example, the Description parameter of New-Alias and its value are enclosed in brackets because they are both optional.

```
[-Description <string>]
```

The brackets also indicate that the Name parameter value (<string>) is required, but the parameter name, "Name," is optional.

[-Name] <string>

-- A right and left bracket ([]) appended to a .NET type indicates that the parameter can accept one or multiple values of that type. Enter the values in a comma-separated list.

For example, the Name parameter of the New-Alias cmdlet takes only one string, but the Name parameter of Get-Process can take one or many strings.

New-Alias [-Name] <string>

New-Alias -Name MyAlias

Get-Process [-Name] <string[]>

Get-Process -Name Explorer, Winlogon, Services

-- Braces ({}) indicate an "enumeration," which is a set of valid values for a parameter.

The values in the braces are separated by vertical bars (|). These bars indicate an "exclusive or" choice, meaning that you can choose only one value from the set of values that are listed inside the braces.

For example, the syntax for the New-Alias cmdlet includes the following value enumeration for the Option parameter:

-Option {None | ReadOnly | Constant | Private | AllScope}

The braces and vertical bars indicate that you can choose any one of the listed values for the Option parameter, such as ReadOnly or AllScope.

-Option ReadOnly

Optional Items

Brackets ([]) surround optional items. For example, in the New-Alias cmdlet syntax description, the Scope parameter is optional. This is indicated in the syntax by the brackets around the parameter name and type:

[-Scope <string>]

Both the following examples are correct uses of the New-Alias cmdlet:

```
New-Alias -Name utd -Value Update-TypeData
New-Alias -Name utd -Value Update-TypeData -Scope global
```

A parameter name can be optional even if the value for that parameter is required. This is indicated in the syntax by the brackets around the parameter name but not the parameter type, as in this example from the New-Alias cmdlet:

```
[-Name] <string> [-Value] <string>
```

The following commands correctly use the New-Alias cmdlet. The commands produce the same result.

```
New-Alias -Name utd -Value Update-TypeData
New-Alias -Name utd Update-TypeData
New-Alias utd -Value Update-TypeData
New-Alias utd Update-TypeData
```

If the parameter name is not included in the statement as typed, Windows PowerShell tries to use the position of the arguments to assign the values to parameters.

The following example is not complete:

```
New-Alias utd
```

This cmdlet requires values for both the Name and Value parameters.

In syntax examples, brackets are also used in naming and casting to .NET Framework types. In this context, brackets do not indicate an element is optional.

KEYWORDS

about_Symbols
about_Punctuation
about_Help_Syntax

SEE ALSO

about_Parameters

Get-Command
Get-Help

about_Comment_Based_Help

SHORT DESCRIPTION

Describes how to write comment-based help topics for functions and scripts.

LONG DESCRIPTION

You can write comment-based help topics for functions and scripts by 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 the contents of comment-based help.

You can also write XML-based help files for functions and scripts. To enable the Get-Help cmdlet to find the XML-based help file for a function or script, use the ExternalHelp keyword. Without this keyword, Get-Help cannot find XML-based help topics for functions or scripts.

This topic explains how to write help topics for functions and scripts. For information about how to display help topics for functions and scripts, see Get-Help.

NOTE:

The Update-Help and Save-Help cmdlets work only on XML files. Updatable Help does not support comment-based help topics.

TROUBLESHOOTING NOTE:

Default values and a value for "Accept Wildcard characters" do not appear in the parameter attribute table even when they are defined in the function or script. To help users, provide this information in the parameter description.

SYNTAX FOR COMMENT-BASED HELP

The syntax for comment-based help is as follows:

```
# .< help keyword>
# <help content>
```

-or -

```
<#
   .< help keyword>
```

```
    < help content>
  #>
```

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.

All of the lines in a comment-based help topic must be contiguous. If a comment-based help topic follows a comment that is not 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.

Keywords define each section of comment-based help. Each comment-based help keyword is preceded by a dot (.). The keywords can appear in any order. The keyword names are not case-sensitive.

For example, the Description keyword precedes a description of a function or script.

```
  <#
.Description
     Get-Function displays the name and syntax of all functions in the session.
  #>
```

The comment block must contain at least one 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.

SYNTAX FOR COMMENT-BASED HELP IN FUNCTIONS

Comment-based help for a function can appear in one of three locations:

-- At the beginning of the function body.

-- At the end of the function body.

-- Before the Function keyword. There cannot be more than one blank line between the last line of the function help and the Function keyword.

For example:

```
function Get-Function
{
  <#
  .< help keyword>
  < help content>
  #>

  <function commands>
}
```

-or -

```
function Get-Function
{
  <function commands>

  <#
  .< help keyword>
  < help content>
  #>
}
```

-or -

```
<#
.< help keyword>
< help content>
#>
function Get-Function { }
```

SYNTAX FOR COMMENT-BASED HELP IN SCRIPTS

Comment-based help for a script can appear in one of the following two locations in the script.

-- At the beginning of the script file. Script help can be preceded in the script only by comments and blank lines.

-- 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.

-- At the end of the script file. However, if the script is signed, place Comment-based help at the beginning of the script file. The end of the script is occupied by the signature block.

For example:

```
<#
.< help keyword>
< help content>
#>


function Get-Function { }
```

-or-

```
function Get-Function { }

<#
.< help keyword>
< help content>
#>
```

SYNTAX FOR COMMENT-BASED HELP IN SCRIPT MODULES

In a script module (.psm1), comment-based help uses the syntax for functions, not the syntax for scripts. You cannot use the script syntax to provide help for all functions defined in a script module.

For example, if you are using the ExternalHelp keyword to identify the XML-based help files for the functions in a script module, you must add an ExternalHelp comment to each function.

```
# .ExternalHelp <XML-file-name>
function <function-name>
{
   ...
}
```

COMMENT-BASED HELP KEYWORDS

The following are valid comment-based help keywords. They are 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 are not case-sensitive.

.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. Add a .PARAMETER keyword for each parameter in the function or script syntax.

Type the parameter name on the same line as the .PARAMETER keyword. Type the parameter description on the lines following the .PARAMETER keyword. Windows PowerShell interprets all text between the .PARAMETER line and the next keyword or the end of the comment block as part of the parameter description. The description can include paragraph breaks.

The Parameter keywords can appear in any order in the comment block, but the function or script syntax determines the order in which the parameters (and their descriptions) appear in help topic. To change the order, change the syntax.

You can also specify a parameter description by placing a comment in the function or script syntax immediately before the parameter variable name. If you use both a syntax comment and a Parameter keyword, the description associated with the Parameter keyword is used, and the syntax 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 Microsoft .NET Framework types of objects that can be piped to the function or script. You can also include a description of the input objects.

.OUTPUTS
The .NET Framework type of the objects that the cmdlet returns. You can also include a description of the returned objects.

.NOTES
Additional information about the function or script.

.LINK
The name of a related topic. The value appears on the line below the .LINE keyword and must be preceded by a comment symbol (#) or included in the comment block.

Repeat the .LINK 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 technology or feature that the function or script uses, or to which it is related. This content appears when the Get-Help command includes the Component parameter of Get-Help.

.ROLE
The user role for the help topic. This content appears when the Get-Help command includes the Role parameter of Get-Help.

.FUNCTIONALITY
The intended use of the function. This content appears when the Get-Help command includes the Functionality parameter of Get-Help.

.FORWARDHELPTARGETNAME <Command-Name>
Redirects to the help topic for the specified command. You can redirect users to any help topic, including help topics for a function, script, cmdlet, or provider.

.FORWARDHELPCATEGORY <Category>
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.

.REMOTEHELPRUNSPACE <PSSession-variable>
Specifies a session that contains the help topic. Enter a variable that contains a PSSession. This keyword is used by the Export-PSSession cmdlet to find the help topics for the exported commands.

.EXTERNALHELP <XML Help File>
Specifies an XML-based help file for the script or function.

The ExternalHelp keyword is required when a function or script is documented in XML files. Without this keyword, Get-Help cannot 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 does not display

comment-based help, even if it cannot 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 file name 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:
<ScriptModule.psm1>-help.xml

If the function is not 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 Create Cmdlet Help" in the MSDN (Microsoft Developer Network)
library at http://go.microsoft.com/fwlink/?LinkID=123415.

AUTOGENERATED CONTENT
The name, syntax, parameter list, parameter attribute table, common
parameters, and remarks are automatically generated by the Get-Help cmdlet.

Name:
The Name section of a function help topic is taken from the function
name in the function syntax. The Name of a script help topic is
taken from the script file name. To change the name or its
capitalization, change the function syntax or the script file name.

Syntax:
The Syntax section of the help topic is generated from the function
or script syntax. To add detail to the help topic syntax, such as
the .NET Framework type of a parameter, add the detail to the syntax.
If you do not specify a parameter type, the "Object" type is
inserted as the default value.

Parameter List:
The Parameter list in the help topic is generated from the function
or script syntax and from the descriptions that you add by using the
Parameters keyword. The function parameters appear in the "Parameters"
section of the help topic in the same order that they appear in
the function or script syntax. The spelling and capitalization of
parameter names is also taken from the syntax; it is not 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.

TROUBLESHOOTING NOTE:
Default values do not appear in the parameter attribute table, even when they are defined in the function or script. To help users, list the default value in the parameter description.

Remarks:
The Remarks section of the help topic is automatically generated from the function or script name. You cannot change or affect its content.

DISABLING COMMENT-BASED HELP

[This technique was suggested by Rich Prescott, a Windows engineer from New York, NY.]

You can disable comment-based help. This makes the comment-based help ineffective without deleting it.

To disable comment-based help, enclose the comments in a here-string. To hide the here-string, assign it to a variable or pipe it to the Out-Null cmdlet.

While it is disabled, the comment-based help has no effect.

For example, the following function has comment-based help.

```
<#
.SYNOPSIS
Adds a file name extension to a supplied name.
#>
function Add-Extension
{
   param ([string]$Name,[string]$Extension = "txt")
   ...
}
```

To disable the comment-based help, enclose it in a here-string,

as shown in the following example.

```
@"
<#
.SYNOPSIS
Adds a file name extension to a supplied name.
#>
"@
function Add-Extension
{
   param ([string]$Name,[string]$Extension = "txt")
   ...
}
```

To hide the disabled comment-based help, assign the here-string to a local variable that is not otherwise used in the function, as shown in the following example. You can also pipe it to the Out-Null cmdlet.

```
$x = @"
<#
.SYNOPSIS
Adds a file name extension to a supplied name.
#>
"@
function Add-Extension
{
   param ([string]$Name,[string]$Extension = "txt")
   ...
}
```

For more information about here-strings, see about_Quoting_Rules (http://go.microsoft.com/fwlink/?LinkID=113253).

EXAMPLES

Example 1: Comment-based Help for a Function

The following sample function includes comment-based help:

```
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 cannot pipe objects to Add-Extension.

.OUTPUTS
System.String. Add-Extension returns a string with the extension or file name.

.EXAMPLE
C:\PS> extension -name "File"
File.txt

.EXAMPLE
C:\PS> extension -name "File" -extension "doc"
File.doc

.EXAMPLE
C:\PS> extension "File" "doc"
File.doc

.LINK
http://www.fabrikam.com/extension.html

.LINK
Set-Item
#>
}
```

The results are as follows:

```
C:\PS> get-help add-extension -full

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 cannot pipe objects to Add-Extension.

OUTPUTS
System.String. Add-Extension returns a string with the extension or file name.

---------- EXAMPLE 1 ----------

C:\PS> extension -name "File"
File.txt

---------- EXAMPLE 2 ----------

```
    C:\PS> extension -name "File" -extension "doc"
    File.doc

    -------------------------- EXAMPLE 3 --------------------------

    C:\PS> extension "File" "doc"
    File.doc

  RELATED LINKS
    http://www.fabrikam.com/extension.html
    Set-Item
```

Example 2: Parameter Descriptions in Function Syntax

This example is the same as the previous one, except that the parameter descriptions are inserted in the function syntax. This format is most useful when the descriptions are brief.

```
  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.

    .DESCRIPTION
    Adds a file name extension to a supplied name. Takes any strings for the file name or extension.

    .INPUTS
    None. You cannot pipe objects to Add-Extension.

    .OUTPUTS
    System.String. Add-Extension returns a string with the extension or file name.
```

```
    .EXAMPLE
    C:\PS> extension -name "File"
    File.txt

    .EXAMPLE
    C:\PS> extension -name "File" -extension "doc"
    File.doc

    .EXAMPLE
    C:\PS> extension "File" "doc"
    File.doc

    .LINK
    http://www.fabrikam.com/extension.html

    .LINK
    Set-Item
    #>
}
```

Example 3: Comment-based Help for a Script

The following sample script includes comment-based help.

Notice the blank lines between the closing "#>" and the Param statement. In a script that does not 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, not the script.

```
  <#
  .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 cannot pipe objects to Update-Month.ps1.

.OUTPUTS
None. Update-Month.ps1 does not generate any output.

.EXAMPLE
C:\PS> .\Update-Month.ps1

.EXAMPLE
C:\PS> .\Update-Month.ps1 -inputpath C:\Data\January.csv

.EXAMPLE
C:\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 is not in a directory that is listed in the Path environment variable, the Get-Help command that gets the script help must specify the script path.

```
PS C:\ps-test> get-help .\update-month.ps1 -full

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 cannot pipe objects to Update-Month.ps1.

OUTPUTS
None. Update-Month.ps1 does not generate any output.

---------- EXAMPLE 1 ----------

C:\PS> .\Update-Month.ps1

---------- EXAMPLE 2 ----------

C:\PS> .\Update-Month.ps1 -inputpath C:\Data\January.csv

---------- EXAMPLE 3 ----------

C:\PS> .\Update-Month.ps1 -inputpath C:\Data\January.csv -outputPath C:\Reports\2009\January.csv

RELATED LINKS

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 for Updatable Help and to provide help topics in 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.

Note that the value of the ExternalHelp keyword appears on the same line as the keyword. Any other placement is ineffective.

```
#  .ExternalHelp C:\MyScripts\Update-Month-Help.xml

param ([string]$InputPath, [string]$OutPutPath)
function Get-Data { }
...
```

The following example shows three valid placements of the ExternalHelp keyword in a function.

```
function Add-Extension
{
   # .ExternalHelp C:\ps-test\Add-Extension.xml

   param ([string] $name, [string]$extension = "txt")
   $name = $name + "." + $extension
   $name
}

function Add-Extension
{
   param ([string] $name, [string]$extension = "txt")
   $name = $name + "." + $extension
   $name

   # .ExternalHelp C:\ps-test\Add-Extension.xml
}

# .ExternalHelp C:\ps-test\Add-Extension.xml
function Add-Extension
{
   param ([string] $name, [string]$extension = "txt")
```

```
        $name = $name + "." + $extension
        $name
    }
```

Example 5: Redirecting to a Different Help Topic

The following code is an excerpt from the beginning of the built-in help function in Windows 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.

```
    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:

```
    C:\PS> get-help help

    NAME
        Get-Help

    SYNOPSIS
        Displays information about Windows PowerShell cmdlets and concepts.
    ...

KEYWORDS
    about_Comment-Based_Help

SEE ALSO
    about_Functions
    about_Functions_Advanced_Parameters
    about_Scripts
```

"How to Write Cmdlet Help" (http://go.microsoft.com/fwlink/?LinkID=123415)

about_CommonParameters

SHORT DESCRIPTION

Describes the parameters that can be used with any cmdlet.

LONG DESCRIPTION

The common parameters are a set of cmdlet parameters that you can use with any cmdlet. They are implemented by Windows PowerShell, not by the cmdlet developer, and they are automatically available to any cmdlet.

You can use the common parameters with any cmdlet, but they might not have an effect on all cmdlets. For example, if a cmdlet does not generate any verbose output, using the Verbose common parameter has no effect.

The common parameters are also available on advanced functions that use the CmdletBinding attribute or the Parameter attribute, and on all workflows.

Several common parameters override system defaults or preferences that you set by using the Windows PowerShell preference variables. Unlike the preference variables, the common parameters affect only the commands in which they are used.

In addition to the common parameters, many cmdlets offer the WhatIf and Confirm risk mitigation parameters. Cmdlets that involve risk to the system or to user data usually offer these parameters.

The following list displays the common parameters. Their aliases are listed in parentheses.

-Debug (db)
-ErrorAction (ea)
-ErrorVariable (ev)
-InformationAction
-InformationVariable
-OutVariable (ov)
-OutBuffer (ob)
-PipelineVariable (pv)
-Verbose (vb)
-WarningAction (wa)
-WarningVariable (wv

The risk mitigation parameters are:

-WhatIf (wi)
-Confirm (cf)

For more information about preference variables, type:
help about_Preference_Variables

COMMON PARAMETER DESCRIPTIONS

-Debug[:{$true | $false}]
Alias: db

Displays programmer-level detail about the operation performed by the command. This parameter works only when the command generates a debugging message. For example, this parameter works when a command contains the Write-Debug cmdlet.

The Debug parameter overrides the value of the $DebugPreference variable for the current command, setting the value of $DebugPreference to Inquire. Because the default value of the $DebugPreference variable is SilentlyContinue, debugging messages are not displayed by default.

Valid values:

$true (-Debug:$true). Has the same effect as -Debug.

$false (-Debug:$false). Suppresses the display of debugging messages when the value of the $DebugPreference is not SilentlyContinue (the default).

-ErrorAction[:{Continue | Ignore | Inquire | SilentlyContinue | Stop | Suspend }]
Alias: ea

Determines how the cmdlet responds to a non-terminating error from the command. This parameter works only when the command generates a non-terminating error, such as those from the Write-Error cmdlet.

The ErrorAction parameter overrides the value of the $ErrorActionPreference variable for the current command. Because the default value of the $ErrorActionPreference variable is Continue, error messages are displayed and execution continues unless you use the ErrorAction parameter.

The ErrorAction parameter has no effect on terminating errors (such as missing data, parameters that are not valid, or insufficient permissions) that prevent a command from completing successfully.

Valid values:

Continue. Displays the error message and continues executing the command. "Continue" is the default value.

Ignore. Suppresses the error message and continues executing the command. Unlike SilentlyContinue, Ignore does not add the error message to the $Error automatic variable. The Ignore value is introduced in Windows PowerShell 3.0.

Inquire. Displays the error message and prompts you for confirmation before continuing execution. This value is rarely used.

SilentlyContinue. Suppresses the error message and continues executing the command.

Stop. Displays the error message and stops executing the command.

Suspend. This value is only available in Windows PowerShell workflows. When a workflow runs into terminating error, this action preference automatically suspends the job to allow for further investigation. After investigation, the workflow can be resumed.

-ErrorVariable [+]<variable-name>
Alias: ev

Stores error messages about the command in the specified variable and in the $Error automatic variable. For more information, type the following command:

get-help about_Automatic_Variables

By default, new error messages overwrite error messages that are already stored in the variable. To append the error message to the variable content, type a plus sign (+) before the variable name.

For example, the following command creates the $a variable and then stores any errors in it:

Get-Process -Id 6 -ErrorVariable a

The following command adds any error messages to the $a variable:

Get-Process -Id 2 -ErrorVariable +a

The following command displays the contents of $a:

$a

You can use this parameter to create a variable that contains only error messages from specific commands. The $Error automatic variable contains error messages from all the commands in the session. You can use array notation, such as $a[0] or $error[1,2] to refer to specific errors stored in the variables.

-InformationAction [:{SilentlyContinue | Stop | Continue | Inquire | Ignore | Suspend}]
Alias: ia

Introduced in Windows PowerShell 5.0. Within the command or script in which it is used, the InformationAction common parameter overrides the value of the $InformationPreference preference variable, which by default is set to SilentlyContinue. When you use Write-Information in a script with -InformationAction, Write-Information values are shown depending on the value of the -InformationAction parameter. For more information about $InformationPreference, see about_Preference_Variables.

Valid values:

Stop: Stops a command or script at an occurrence of the Write-Information command.

Ignore Suppresses the informational message and continues running the command. Unlike SilentlyContinue, Ignore completely "forgets" the informational message; it does not add the informational message to the information stream.

Inquire: Displays the informational message that you specify in a Write-Information command, then asks whether you want to continue.

Continue: Displays the informational message, and continues running.

Suspend: Automatically suspends a workflow job after a Write-Information command is carried out, to allow users to see the messages before continuing. The workflow can be resumed at the user's discretion.

SilentlyContinue: (Default) No effect. The informational messages are not displayed, and the script continues without

interruption.

-InformationVariable [+]<variable-name>
Alias: iv

Introduced in Windows PowerShell 5.0. Within the command or script in which it is used, the InformationVariable common parameter stores in a variable a string that you specify by adding the Write-Information command. Write-Information values are shown depending on the value of the -InformationAction common parameter; if you do not add the -InformationAction common parameter, Write-Information strings are shown depending on the value of the $InformationPreference preference variable. For more information about $InformationPreference, see about_Preference_Variables.

-OutBuffer <Int32>
Alias: ob

Determines the number of objects to accumulate in a buffer before any objects are sent through the pipeline. If you omit this parameter, objects are sent as they are generated.

This resource management parameter is designed for advanced users. When you use this parameter, Windows PowerShell does not call the next cmdlet in the pipeline until the number of objects generated equals OutBuffer + 1. Thereafter, it sends all objects as they are generated.

-OutVariable [+]<variable-name>
Alias: ov

Stores output objects from the command in the specified variable and displays it at the command line.

To add the output to the variable, instead of replacing any output that might already be stored there, type a plus sign (+) before the variable name.

For example, the following command creates the $out variable and stores the process object in it:

```
Get-Process PowerShell -OutVariable out
```

The following command adds the process object to the $out variable:

```
Get-Process iexplore -OutVariable +out
```

The following command displays the contents of the $out variable:

```
$out
```

-PipelineVariable <String>
Alias: pv

PipelineVariable stores the value of the current pipeline element as a variable, for any named command as it flows through the pipeline.

Valid values are strings, the same as for any variable names.

The following is an example of how PipelineVariable works. In this example, the PipelineVariable parameter is added to a Foreach-Object command to store the results of the command in variables. A range of numbers, 1 to 10, are piped into the first Foreach-Object command, the results of which are stored in a variable named Left.

The results of the first Foreach-Object command are piped into a second Foreach-Object command, which filters the objects returned by the first Foreach-Object command. The results of the second command are stored in a variable named Right.

In the third Foreach-Object command, the results of the first two Foreach-Object piped commands, represented by the variables Left and Right, are processed by using a multiplication operator. The command instructs objects stored in the Left and Right variables to be multiplied, and specifies that the results should be displayed as "Left range member * Right range member = product".

```
1..10 | Foreach-Object -PipelineVariable Left -Process { $_ } |
 >>    Foreach-Object -PV Right -Process { 1..10 } |
 >>    Foreach-Object -Process { "$Left * $Right = " + ($Left * $Right) }
 >>
 1 * 1 = 1
 1 * 2 = 2
 1 * 3 = 3
 1 * 4 = 4
 1 * 5 = 5
```

-Verbose[:{$true | $false}]

Alias: vb

Displays detailed information about the operation performed by the command. This information resembles the information in a trace or in a transaction log. This parameter works only when the command generates a verbose message. For example, this parameter works when a command contains the Write-Verbose cmdlet.

The Verbose parameter overrides the value of the $VerbosePreference variable for the current command. Because the default value of the $VerbosePreference variable is SilentlyContinue, verbose messages are not displayed by default.

Valid values:

$true (-Verbose:$true) has the same effect as -Verbose.

$false (-Verbose:$false) suppresses the display of verbose messages. Use this parameter when the value of $VerbosePreference is not SilentlyContinue (the default).

-WarningAction[:{Continue | Inquire | SilentlyContinue | Stop}]
Alias: wa

Determines how the cmdlet responds to a warning from the command. "Continue" is the default value. This parameter works only when the command generates a warning message. For example, this parameter works when a command contains the Write-Warning cmdlet.

The WarningAction parameter overrides the value of the $WarningPreference variable for the current command. Because the default value of the $WarningPreference variable is Continue, warnings are displayed and execution continues unless you use the WarningAction parameter.

Valid Values:

Continue. Displays the warning message and continues executing the command. "Continue" is the default value.

Inquire. Displays the warning message and prompts you for confirmation before continuing execution. This value is rarely used.

SilentlyContinue. Suppresses the warning message and continues executing the command.

Stop. Displays the warning message and stops executing the command.

NOTE: The WarningAction parameter does not override the value of the $WarningAction preference variable when the parameter is used in a command to run a script or function.

-WarningVariable [+]<variable-name>
Alias: wv

Stores warnings about the command in the specified variable.

All generated warnings are saved in the variable even if the warnings are not displayed to the user.

To append the warnings to the variable content, instead of replacing any warnings that might already be stored there, type a plus sign (+) before the variable name.

For example, the following command creates the $a variable and then stores any warnings in it:

Get-Process -Id 6 -WarningVariable a

The following command adds any warnings to the $a variable:

Get-Process -Id 2 -WarningVariable +a

The following command displays the contents of $a:

$a

You can use this parameter to create a variable that contains only warnings from specific commands. You can use array notation, such as $a[0] or $warning[1,2] to refer to specific warnings stored in the variable.

NOTE: The WarningVariable parameter does not capture warnings from nested calls in functions or scripts.

Risk Management Parameter Descriptions

-WhatIf[:{$true | $false}]
Alias: wi

Displays a message that describes the effect of the command, instead of executing the command.

The WhatIf parameter overrides the value of the $WhatIfPreference variable for the current command. Because the default value of the $WhatIfPreference variable is 0 (disabled), WhatIf behavior is not performed without the WhatIf parameter. For more information, type the following command:

Get-Help about_Preference_Variables

Valid values:

$true (-WhatIf:$true). Has the same effect as -WhatIf.

$false (-WhatIf:$false). Suppresses the automatic WhatIf behavior that results when the value of the $WhatIfPreference variable is 1.

For example, the following command uses the WhatIf parameter in a Remove-Item command:

PS> Remove-Item Date.csv -WhatIf

Instead of removing the item, Windows PowerShell lists the operations it would perform and the items that would be affected. This command produces the following output:

What if: Performing operation "Remove File" on
Target "C:\ps-test\date.csv".

-Confirm[:{$true | $false}]
Alias: cf

Prompts you for confirmation before executing the command.

The Confirm parameter overrides the value of the $ConfirmPreference variable for the current command. The default value is High. For more information, type the following command:

Get-Help about_Preference_Variables

Valid values:

$true (-Confirm:$true). Has the same effect as -Confirm.

$false(-Confirm:$false). Suppresses automatic confirmation, which occurs when the value of $ConfirmPreference is less than or equal to the estimated risk of the cmdlet.

For example, the following command uses the Confirm parameter with a Remove-Item command. Before removing the item, Windows PowerShell lists the operations it would perform and the items that would be affected, and asks for approval.

```
PS C:\ps-test> Remove-Item tmp*.txt -Confirm
```

This command produces the following output:

```
Confirm
Are you sure you want to perform this action?
Performing operation "Remove File" on Target " C:\ps-test\tmp1.txt
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend
[?] Help (default is "Y"):
```

The Confirm response options are as follows:

```
Yes (Y)        Perform the action.
Yes to All (A)  Perform all actions and suppress subsequent
            Confirm queries for this command.
No (N):        Do not perform the action.
No to All (L):  Do not perform any actions and suppress subsequent
            Confirm queries for this command.
Suspend (S):    Pause the command and create a temporary session.
Help (?)       Display help for these options.
```

The Suspend option places the command on hold and creates a temporary nested session in which you can work until you're ready to choose a Confirm option. The command prompt for the nested session has two extra carets (>>) to indicate that it's a child operation of the original parent command. You can run commands and scripts in the nested session. To end the nested session and return to the Confirm options for the original command, type "exit".

In the following example, the Suspend option (S) is used to halt a command temporarily while the user checks the help for a command parameter. After obtaining the needed information, the user types "exit" to end the nested prompt and then selects the Yes (y) response to the Confirm query.

```
PS C:\ps-test> New-Item -ItemType File -Name Test.txt -Confirm

Confirm
Are you sure you want to perform this action?
Performing operation "Create File" on Target "Destination: C:\ps-test\test.txt".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): s

PS C:\ps-test>>> Get-Help New-Item -Parameter ItemType

-ItemType <string>
   Specifies the provider-specified type of the new item.

   Required?                    false
   Position?                    named
   Default value
   Accept pipeline input?       true (ByPropertyName)
   Accept wildcard characters?  false


PS C:\ps-test>>> exit

Confirm
Are you sure you want to perform this action?
Performing operation "Create File" on Target "Destination: C:\ps-test\test.txt".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y


   Directory: C:\ps-test


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---        8/27/2010   2:41 PM          0 test.txt



KEYWORDS
   about_Common_Parameters

SEE ALSO
   about_Preference_Variables
   Write-Debug
   Write-Warning
   Write-Error
   Write-Verbose
```

about_Comparison_Operators

SHORT DESCRIPTION

Describes the operators that compare values in Windows PowerShell.

LONG DESCRIPTION

Comparison operators let you specify conditions for comparing values and finding values that match specified patterns. To use a comparison operator, specify the values that you want to compare together with an operator that separates these values.

Windows PowerShell includes the following comparison operators:

-eq
-ne
-gt
-ge
-lt
-le
-Like
-NotLike
-Match
-NotMatch
-Contains
-NotContains
-In
-NotIn
-Replace

By default, all comparison operators are case-insensitive. To make a comparison operator case-sensitive, precede the operator name with a "c". For example, the case-sensitive version of "-eq" is "-ceq". To make the case-insensitivity explicit, precede the operator with an "i". For example, the explicitly case-insensitive version of "-eq" is "-ieq".

When the input to an operator is a scalar value, comparison operators return a Boolean value. When the input is a collection of values, the comparison operators return any matching values. If there are no matches in a collection, comparison operators do not return anything.

The exceptions are the containment operators (-Contains, -NotContains), the In operators (-In, -NotIn), and the type operators (-Is, -IsNot), which always return a Boolean value.

Windows PowerShell supports the following comparison operators.

-eq

Description: Equal to. Includes an identical value.
Example:

```
PS C:\> "abc" -eq "abc"
True

PS C:\> "abc" -eq "abc", "def"
False

PS C:\> "abc", "def" -eq "abc"
abc
```

-ne
Description: Not equal to. Includes a different value.
Example:

```
PS C:\> "abc" -ne "def"
True

PS C:\> "abc" -ne "abc"
False

PS C:\> "abc" -ne "abc", "def"
True

PS C:\> "abc", "def" -ne "abc"
def
```

-gt
Description: Greater-than.
Example:

```
PS C:\> 8 -gt 6
True

PS C:\> 7, 8, 9 -gt 8
9
```

-ge
Description: Greater-than or equal to.
Example:

```
PS C:\> 8 -ge 8
True
```

```
PS C:\> 7, 8, 9 -ge 8
8
9
```

-lt
Description: Less-than.
Example:

```
PS C:\> 8 -lt 6
False

PS C:\> 7, 8, 9 -lt 8
7
```

-le
Description: Less-than or equal to.
Example:

```
PS C:\> 6 -le 8
True

PS C:\> 7, 8, 9 -le 8
7
8
```

-Like
Description: Match using the wildcard character (*).
Example:

```
PS C:\> "Windows PowerShell" -like "*shell"
True

PS C:\> "Windows PowerShell", "Server" -like "*shell"
Windows PowerShell
```

-NotLike
Description: Does not match using the wildcard character (*).
Example:

```
PS C:\> "Windows PowerShell" -NotLike "*shell"
False

PS C:\> "Windows PowerShell", "Server" -NotLike "*shell"
Server
```

-Match
Description: Matches a string using regular expressions.
When the input is scalar, it populates the
$Matches automatic variable.
Example:

```
PS C:\> "Sunday" -Match "sun"
True

PS C:\> $matches
Name Value
---- -----
0    Sun

PS C:\> "Sunday", "Monday" -Match "sun"
Sunday
```

-NotMatch
Description: Does not match a string. Uses regular expressions.
When the input is scalar, it populates the $Matches
automatic variable.
Example:

```
PS C:\> "Sunday" -NotMatch "sun"
False

PS C:\> $matches
Name Value
---- -----
0    sun

PS C:\> "Sunday", "Monday" -NotMatch "sun"
Monday
```

-Contains
Description: Containment operator. Tells whether a collection of reference values includes a single test value. Always returns a Boolean value. Returns TRUE only when the test value exactly matches at least one of the reference values.

When the test value is a collection, the Contains operator uses reference equality. It returns TRUE only when one of the reference values is the same instance of the test value object.

Syntax:

<Reference-values> -Contains <Test-value>

Examples:

```
PS C:\> "abc", "def" -Contains "def"
True

PS C:\> "Windows", "PowerShell" -Contains "Shell"
False  #Not an exact match


# Does the list of computers in $domainServers
# include $thisComputer?
# ----------------------------------------
PS C:\> $domainServers -Contains $thisComputer
True

PS C:\> "abc", "def", "ghi" -Contains "abc", "def"
False

PS C:\> $a = "abc", "def"
PS C:\> "abc", "def", "ghi" -Contains $a
False
PS C:\> $a, "ghi" -Contains $a
True
```

-NotContains

Description: Containment operator. Tells whether a collection of reference values includes a single test value. Always returns a Boolean value. Returns TRUE when the test value is not an exact matches for at least one of the reference values.

When the test value is a collection, the NotContains operator uses reference equality.

Syntax:

<Reference-values> -NotContains <Test-value>

Examples:

```
PS C:\> "Windows", "PowerShell" -NotContains "Shell"
True  #Not an exact match


# Get cmdlet parameters, but exclude common parameters
function get-parms ($cmdlet)
{
```

```
    $Common = "Verbose", "Debug", "WarningAction", "WarningVariable", `
          "ErrorAction", "ErrorVariable", "OutVariable", "OutBuffer"

    $allparms = (Get-Command $Cmdlet).parametersets | foreach {$_.Parameters} | `
          foreach {$_.Name} | Sort-Object | Get-Unique

    $allparms | where {$Common -NotContains $_ }
  }


  # Find unapproved verbs in the functions in my module
  # ---------------------------------------------
  PS C:\> $ApprovedVerbs = Get-Verb | foreach {$_.verb}
  PS C:\> $myVerbs = Get-Command -Module MyModule | foreach {$_.verb}

  PS C:\> $myVerbs | where {$ApprovedVerbs -NotContains $_}
  ForEach
  Sort
  Tee
  Where


-In
 Description: In operator. Tells whether a test value appears in a collection
 of reference values. Always return as Boolean value. Returns TRUE only when
 the test value exactly matches at least one of the reference values.

 When the test value is a collection, the In operator uses reference equality.
 It returns TRUE only when one of the reference values is the same
 instance of the test value object.

 The In operator was introduced in Windows PowerShell 3.0.

 Syntax:
   <Test-value> -in <Reference-values>


 Examples:

   PS C:\> "def" -in "abc", "def"
   True

   PS C:\> "Shell" -in "Windows", "PowerShell"
   False  #Not an exact match

   PS C:\> "Windows" -in "Windows", "PowerShell"
   True  #An exact match
```

```
PS C:\> "Windows", "PowerShell" -in "Windows", "PowerShell", "ServerManager"
False  #Using reference equality

PS C:\> $a = "Windows", "PowerShell"
PS C:\> $a -in $a, "ServerManager"
True  #Using reference equality

# Does the list of computers in $domainServers
# include $thisComputer?
# ---------------------------------------
PS C:\> $thisComputer -in  $domainServers
True
```

-NotIn

Description: NotIn operator. Tells whether a test value appears in a collection of reference values. Always returns a Boolean value. Returns TRUE when the test value is not an exact match for at least one of the reference values.

When the test value is a collection, the In operator uses reference equality. It returns TRUE only when one of the reference values is the same instance of the test value object.

The NotIn operator was introduced in Windows PowerShell 3.0.

Syntax:

```
<Test-value> -NotIn <Reference-values>
```

Examples:

```
PS C:\> "def" -NotIn "abc", "def"
False

PS C:\> "ghi" -NotIn "abc", "def"
True

PS C:\> "Shell" -NotIn "Windows", "PowerShell"
True  #Not an exact match

PS C:\> "Windows" -NotIn "Windows", "PowerShell"
False  #An exact match



# Find unapproved verbs in the functions in my module
# ---------------------------------------
PS C:\> $ApprovedVerbs = Get-Verb | foreach {$_.verb}
```

```
PS C:\> $myVerbs = Get-Command -Module MyModule | foreach {$_.verb}

PS C:\> $myVerbs | where {$_ -NotIn $ApprovedVerbs}
ForEach
Sort
Tee
Where
```

-Replace
Description: Replace operator. Changes the specified elements of a value.
Example:

```
PS C:\> "Get-Process" -Replace "Get", "Stop"
Stop-Process

# Change all .GIF file name extension to .JPG
PS C:\> dir *.gif | foreach {$_ -Replace ".gif", ".jpg"}
```

Equality Operators
The equality operators (-eq, -ne) return a value of TRUE or the matches when one or more of the input values is identical to the specified pattern. The entire pattern must match an entire value.

```
C:PS> 2 -eq 2
True

C:PS> 2 -eq 3
False

C:PS> 1,2,3 -eq 2
2

C:PS> "PowerShell" -eq "Shell"
False

C:PS> "Windows", "PowerShell" -eq "Shell"
C:PS>

PS C:\> "abc", "def", "123" -eq "def"
def
```

```
PS C:\> "abc", "def", "123" -ne "def"
abc
123
```

Containment Operators

The containment operators (-Contains and -NotContains) are similar to the equality operators. However, the containment operators always return a Boolean value, even when the input is a collection.

Also, unlike the equality operators, the containment operators return a value as soon as they detect the first match. The equality operators evaluate all input and then return all the matches in the collection.
The following examples show the effect of the -Contains operator:

```
C:PS> 1,2,3 -contains 2
True

C:PS> "PowerShell" -contains "Shell"
False

C:PS> "Windows", "PowerShell" -contains "Shell"
False

PS C:\> "abc", "def", "123" -contains "def"
True

PS C:\> "true", "blue", "six" -contains "true"
True
```

The following example shows how the containment operators differ from the equal to operator. The containment operators return a value of TRUE on the first match.

```
PS C:\> 1,2,3,4,5,4,3,2,1 -eq 2
2
2

PS C:\> 1,2,3,4,5,4,3,2,1 -contains 2
True
```

In a very large collection, the -Contains operator returns results quicker than the equal to operator.

Match Operators

The match operators (-Match and -NotMatch) find elements that match or do not match a specified pattern using regular expressions.

The syntax is:

```
<string[]> -Match <regular-expression>
<string[]> -NotMatch <regular-expression>
```

The following examples show some uses of the -Match operator:

```
PS C:\> "Windows", "PowerShell" -Match ".shell"
PowerShell

PS C:\> (Get-Command Get-Member -Syntax) -Match "-view"
True

PS C:\> (Get-Command Get-Member -Syntax) -NotMatch "-path"
True

PS C:\> (Get-Content Servers.txt) -Match "^Server\d\d"
Server01
Server02
```

The match operators search only in strings. They cannot search in arrays of integers or other objects.

The -Match and -NotMatch operators populate the $Matches automatic variable when the input (the left-side argument) to the operator is a single scalar object. When the input is scalar, the -Match and -NotMatch operators return a Boolean value and set the value of the $Matches automatic variable to the matched components of the argument.

If the input is a collection, the -Match and -NotMatch operators return the matching members of that collection, but the operator does not populate the $Matches variable.

For example, the following command submits a collection of strings to the -Match operator. The -Match operator returns the items in the collection that match. It does not populate the $Matches automatic variable.

```
PS C:\> "Sunday", "Monday", "Tuesday" -Match "sun"
Sunday

PS C:\> $matches
```

```
PS C:\>
```

In contrast, the following command submits a single string to the -Match operator. The -Match operator returns a Boolean value and populates the $Matches automatic variable.

```
PS C:\> "Sunday" -Match "sun"
True

PS C:\> $matches

Name                           Value
----                           -----
0                              Sun
```

The -NotMatch operator populates the $Matches automatic variable when the input is scalar and the result is False, that it, when it detects a match.

```
PS C:\> "Sunday" -NotMatch "rain"
True

PS C:\> $matches
PS C:\>

PS C:\> "Sunday" -NotMatch "day"
False

PS C:\> $matches
PS C:\>

Name                           Value
----                           -----
0                              day
```

Replace Operator

The -Replace operator replaces all or part of a value with the specified value using regular expressions. You can use the -Replace operator for many administrative tasks, such as renaming files. For example, the following command changes the file name extensions of all .gif files to .jpg:

```
Get-ChildItem | Rename-Item -NewName { $_ -Replace '.gif$','.jpg$' }
```

The syntax of the -Replace operator is as follows, where the <original> placeholder represents the characters to be replaced, and the <substitute> placeholder represents the characters that will replace them:

```
<input> <operator> <original>, <substitute>
```

By default, the -Replace operator is case-insensitive. To make it case sensitive, use -cReplace. To make it explicitly case-insensitive, use -iReplace. Consider the following examples:

```
PS C:\> "book" -Replace "B", "C"
Cook
PS C:\> "book" -iReplace "B", "C"
Cook
PS C:\> "book" -cReplace "B", "C"
book

PS C:\> '<command:parameter required="false" variableLength="true" globbing="false"'`
      | foreach {$_ -replace 'globbing="false"', 'globbing="true"'}
<command:parameter required="false" variableLength="true" globbing="true"
```

Bitwise Operators

Windows PowerShell supports the standard bitwise operators, including bitwise-AND (-bAnd), the inclusive and exclusive bitwise-OR operators (-bOr and -bXor), and bitwise-NOT (-bNot).

Beginning in Windows PowerShell 2.0, all bitwise operators work with 64-bit integers.

Beginning in Windows PowerShell 3.0, the -shr (shift-right) and -shl (shift-left) are introduced to support bitwise arithmetic in Windows PowerShell.

Windows PowerShell supports the following bitwise operators.

Operator	Description	Example
-bAnd	Bitwise AND	PS C:\> 10 -band 3 2

-bOr Bitwise OR (inclusive) PS C:\> 10 -bor 3
11

-bXor Bitwise OR (exclusive) PS C:\> 10 -bxor 3
9

-bNot Bitwise NOT PS C:\> -bNot 10
-11

-shl Shift-left PS C:\> 100 -shl 2
400

-shr Shift-right PS C:\> 100 -shr 1
50

Bitwise operators act on the binary format of a value. For example, the bit structure for the number 10 is 00001010 (based on 1 byte), and the bit structure for the number 3 is 00000011. When you use a bitwise operator to compare 10 to 3, the individual bits in each byte are compared.

In a bitwise AND operation, the resulting bit is set to 1 only when both input bits are 1.

```
1010     (10)
0011     ( 3)
-------------- bAND
0010     ( 2)
```

In a bitwise OR (inclusive) operation, the resulting bit is set to 1 when either or both input bits are 1. The resulting bit is set to 0 only when both input bits are set to 0.

```
1010     (10)
0011     ( 3)
-------------- bOR (inclusive)
1011     (11)
```

In a bitwise OR (exclusive) operation, the resulting bit is set to 1 only when one input bit is 1.

```
1010     (10)
0011     ( 3)
-------------- bXOR (exclusive)
1001     ( 9)
```

The bitwise NOT operator is a unary operator that produces the binary complement of the value. A bit of 1 is set to 0 and a bit of 0 is set to 1.

For example, the binary complement of 0 is -1, the maximum unsigned integer (0xffffffff), and the binary complement of -1 is 0.

```
PS C:\> -bNOT 10
-11
```

```
0000 0000 0000 1010  (10)
------------------------ bNOT
1111 1111 1111 0101  (-11, xfffffff5)
```

In a bitwise shift-left operation, all bits are moved "n" places to the left, where "n" is the value of the right operand. A zero is inserted in the ones place.

When the left operand is an Integer (32-bit) value, the lower 5 bits of the right operand determine how many bits of the left operand are shifted.

When the left operand is a Long (64-bit) value, the lower 6 bits of the right operand determine how many bits of the left operand are shifted.

```
PS C:\> 21 -shl 1
42
```

```
00010101  (21)
00101010  (42)
```

```
PS C:\> 21 -shl 2
84
```

```
00010101  (21)
```

00101010 (42)
01010100 (84)

In a bitwise shift-right operation, all bits are moved "n" places
to the right, where "n" is specified by the right operand. The
shift-right operator (-shr) inserts a zero in the left-most
place when shifting a positive or unsigned value to the right.

When the left operand is an Integer (32-bit) value, the lower 5 bits
of the right operand determine how many bits of the left operand are
shifted.

When the left operand is a Long (64-bit) value, the lower 6 bits of
the right operand determine how many bits of the left operand are
shifted.

```
PS C:\> 21 -shr 1
10
```

00010101 (21)
00001010 (10)

```
PS C:\> 21 -shr 2
5
```

00010101 (21)
00001010 (10)
00000101 (5)

SEE ALSO
about_Operators
about_Regular_Expressions
about_Wildcards
Compare-Object
Foreach-Object
Where-Object

about_Continue

SHORT DESCRIPTION
Describes how the Continue statement immediately returns the program flow
to the top of a program loop.

LONG DESCRIPTION

In a script, the Continue statement immediately returns the program flow to the top of the innermost loop that is controlled by a For, Foreach, or While statement.

The Continue keyword supports labels. A label is a name you assign to a statement in a script. For information about labels, see about_Break.

In the following example, program flow returns to the top of the While loop if the $ctr variable is equal to 5. As a result, all the numbers between 1 and 10 are displayed except for 5:

```
while ($ctr -lt 10)
    {
     $ctr +=1
     if ($ctr -eq 5) {continue}
     Write-Host $ctr
    }
```

Note that in a For loop, execution continues at the first line in the loop. If the arguments of the For statement test a value that is modified by the For statement, an infinite loop may result.

SEE ALSO

about_Break
about_Comparison_Operators
about_Throw
about_Trap
about_Try_Catch_Finally

about_Core_Commands

SHORT DESCRIPTION

Lists the cmdlets that are designed for use with Windows PowerShell providers.

LONG DESCRIPTION

Windows PowerShell includes a set of cmdlets that are specifically designed to manage the items in the data stores that are exposed by Windows PowerShell providers. You can use these cmdlets in the same ways to manage

all the different types of data that the providers make available to you.
For more information about providers, type "get-help about_providers".

For example, you can use the Get-ChildItem cmdlet to list the files in a file system directory, the keys under a registry key, or the items that are exposed by a provider that you write or download.

The following is a list of the Windows PowerShell cmdlets that are designed for use with providers:

ChildItem cmdlets
- Get-ChildItem

Content cmdlets
- Add-Content
- Clear-Content
- Get-Content
- Set-Content

Item cmdlets
- Clear-Item
- Copy-Item
- Get-Item
- Invoke-Item
- Move-Item
- New-Item
- Remove-Item
- Rename-Item
- Set-Item

ItemProperty cmdlets
- Clear-ItemProperty
- Copy-ItemProperty
- Get-ItemProperty
- Move-ItemProperty
- New-ItemProperty
- Remove-ItemProperty
- Rename-ItemProperty
- Set-ItemProperty

Location cmdlets
Get-Location
Pop-Location
Push-Location
Set-Location

Path cmdlets
Join-Path
Convert-Path
Split-Path
Resolve-Path
Test-Path

PSDrive cmdlets
Get-PSDrive
New-PSDrive
Remove-PSDrive

PSProvider cmdlets
Get-PSProvider

For more information about a cmdlet, type "get-help <cmdlet-name>".

SEE ALSO
about_Providers

about_Data_Sections

SHORT DESCRIPTION
Explains Data sections, which isolate text strings and other read-only data from script logic.

LONG DESCRIPTION
Scripts that are designed for Windows PowerShell can have one or more Data sections that contain only data. You can include one or more Data sections in any script, function, or advanced function. The content of the Data section is restricted to a specified subset of the Windows PowerShell scripting language.

Separating data from code logic makes it easier to identify and manage both logic and data. It lets you have separate string resource files for

text, such as error messages and Help strings. It also isolates the code logic, which facilitates security and validation tests.

In Windows PowerShell, the Data section is used to support script internationalization. You can use Data sections to make it easier to isolate, locate, and process strings that will be translated into many user interface (UI) languages.

The Data section is a Windows PowerShell 2.0 feature. Scripts with Data sections will not run in Windows PowerShell 1.0 without revision.

Syntax

The syntax for a Data section is as follows:

```
DATA [-supportedCommand <cmdlet-name>] {

   <Permitted content>
}
```

The Data keyword is required. It is not case-sensitive.

The permitted content is limited to the following elements:

- All Windows PowerShell operators, except -match
- If, Else, and ElseIf statements
- The following automatic variables: $PsCulture, $PsUICulture, $True, $False, and $Null
- Comments
- Pipelines
- Statements separated by semicolons (;)
- Literals, such as the following:

  a

  1

  1,2,3

```
"Windows PowerShell 2.0"

@( "red", "green", "blue" )

@{ a = 0x1; b = "great"; c ="script" }

[XML] @'
 <p> Hello, World </p>
'@
```

- Cmdlets that are permitted in a Data section. By default, only the ConvertFrom-StringData cmdlet is permitted.

- Cmdlets that you permit in a Data section by using the SupportedCommand parameter.

When you use the ConvertFrom-StringData cmdlet in a Data section, you can enclose the key/value pairs in single-quoted or double-quoted strings or in single-quoted or double-quoted here-strings. However, strings that contain variables and subexpressions must be enclosed in single-quoted strings or in single-quoted here-strings so that the variables are not expanded and the subexpressions are not executable.

SupportedCommand

The SupportedCommand parameter allows you to indicate that a cmdlet or function generates only data. It is designed to allow users to include cmdlets and functions in a data section that they have written or tested.

The value of SupportedCommand is a comma-separated list of one or more cmdlet or function names.

For example, the following data section includes a user-written cmdlet, Format-XML, that formats data in an XML file:

```
DATA -supportedCommand Format-XML
    {
      Format-XML -strings string1, string2, string3
    }
```

Using a Data Section

To use the content of a Data section, assign it to a variable and use variable notation to access the content.

For example, the following data section contains a ConvertFrom-StringData command that converts the here-string into a hash table. The hash table is assigned to the $TextMsgs variable.

The $TextMsgs variable is not part of the data section.

```
$TextMsgs = DATA {
   ConvertFrom-StringData -stringdata @'
     Text001 = Windows 7
     Text002 = Windows Server 2008 R2
'@
}
```

To access the keys and values in hash table in $TextMsgs, use the following commands.

```
$TextMsgs.Text001
$TextMsgs.Text002
```

EXAMPLES

Simple data strings.

```
DATA {
   "Thank you for using my Windows PowerShell Organize.pst script."
   "It is provided free of charge to the community."
   "I appreciate your comments and feedback."
}
```

Strings that include permitted variables.

```
DATA {
   if ($null) {
"To get help for this cmdlet, type get-help new-dictionary."
   }
}
```

A single-quoted here-string that uses the ConvertFrom-StringData cmdlet:

```
DATA {
  ConvertFrom-StringData -stringdata @'
    Text001 = Windows 7
    Text002 = Windows Server 2008 R2
```

```
'@
}
```

A double-quoted here-string that uses the ConvertFrom-StringData cmdlet:

```
DATA  {
 ConvertFrom-StringData -stringdata @"
   Msg1 = To start, press any key.
   Msg2 = To exit, type "quit".
"@
}
```

A data section that includes a user-written cmdlet that generates data:

```
DATA -supportedCommand Format-XML {
      Format-XML -strings string1, string2, string3
    }
```

SEE ALSO

about_Automatic_Variables
about_Comparison_Operators
about_Hash_Tables
about_If
about_Operators
about_Quoting_Rules
about_Script_Internationalization
ConvertFrom-StringData
Import-LocalizedData

about_Debuggers

SHORT DESCRIPTION

Describes the Windows PowerShell debugger.

LONG DESCRIPTION

Debugging is the process of examining a script while it is running to identify and correct errors in the script instructions. The Windows PowerShell debugger can help you examine and identify errors and inefficiencies in your scripts, functions, commands, Windows PowerShell workflows, Windows PowerShell Desired State Configuration (DSC)

configurations, or expressions.

Starting in Windows PowerShell 5.0, the Windows PowerShell debugger has been updated to debug scripts, functions, workflows, commands, configurations, or expressions that are running in either the console or Windows PowerShell ISE on remote computers. You can run Enter-PSSession to start an interactive remote PowerShell session in which you can set breakpoints and debug script files and commands on the remote computer. Enter-PSSession functionality has been updated to let you reconnect to and enter a disconnected session that is running a script or command on a remote computer. If the running script hits a breakpoint, your client session automatically starts the debugger. If the disconnected session that is running a script has already hit a breakpoint, and is stopped at the breakpoint, Enter-PSSession automatically starts the command-line debugger, after you reconnect to the session.

The Windows PowerShell debugger can also be used to debug Windows PowerShell workflows, in either the Windows PowerShell console, or in Windows PowerShell ISE. Starting in Windows PowerShell 5.0, you can debug within running jobs or processes, either locally or remotely.

You can use the features of the Windows PowerShell debugger to examine a Windows PowerShell script, function, command, workflow, or expression while it is running. The Windows PowerShell debugger includes a set of cmdlets that let you set breakpoints, manage breakpoints, and view the call stack.

Debugger Cmdlets

The Windows PowerShell debugger includes the following set of cmdlets:

Set-PsBreakpoint: Sets breakpoints on lines, variables, and commands.

Get-PsBreakpoint: Gets breakpoints in the current session.

Disable-PsBreakpoint: Turns off breakpoints in the current session.

Enable-PsBreakpoint: Re-enables breakpoints in the current session.

Remove-PsBreakpoint: Deletes breakpoints from the current session.

Get-PsCallStack: Displays the current call stack.

Starting and Stopping the Debugger

To start the debugger, set one or more breakpoints. Then, run the script, command, or function that you want to debug.

When you reach a breakpoint, execution stops, and control is turned over to the debugger.

To stop the debugger, run the script, command, or function until it is complete. Or, type "stop" or "t".

Debugger Commands

When you use the debugger in the Windows PowerShell console, use the following commands to control the execution. In Windows PowerShell ISE, use commands on the Debug menu.

Note: For information about how to use the debugger in other host applications, see the host application documentation.

s, Step-into Executes the next statement and then stops.

v, Step-over Executes the next statement, but skips functions and invocations. The skipped statements are executed, but not stepped through.

Ctrl+Break
(Break All in ISE) Breaks into a running script within either the Windows PowerShell console, or Windows PowerShell ISE. Note that Ctrl+Break in Windows PowerShell 2.0, 3.0, and 4.0 closes the program. Break All works on both local and remote interactively-running scripts.

o, Step-out Steps out of the current function; up one level if nested. If in the main body, it continues to the end or the next breakpoint. The skipped statements are executed, but not stepped through.

c, Continue Continues to run until the script is complete or until the next breakpoint is reached. The skipped statements are executed, but not stepped through.

l, List Displays the part of the script that is executing. By default, it displays the current line, five previous lines, and 10 subsequent lines. To continue listing the script, press ENTER.

l <m>, List Displays 16 lines of the script beginning with the line number specified by <m>.

l <m> <n>, List Displays <n> lines of the script, beginning with the line number specified by <m>.

q, Stop, Exit Stops executing the script, and exits the debugger. If you are debugging a job by running the Debug-Job cmdlet, the Exit command detaches the debugger, and allows the job to continue running.

k, Get-PsCallStack Displays the current call stack.

<Enter> Repeats the last command if it was Step (s), Step-over (v), or List (l). Otherwise, represents a submit action.

?, h Displays the debugger command Help.

To exit the debugger, you can use Stop (q).
Starting in Windows PowerShell 5.0, you can run the Exit command to exit a nested debugging session that you started by running either Debug-Job or Debug-Runspace.

By using these debugger commands, you can run a script, stop on a point of concern, examine the values of variables and the state of the system, and continue running the script until you have identified a problem.

NOTE: If you step into a statement with a redirection operator, such as ">", the Windows PowerShell debugger steps over all remaining statements in the script.

Displaying the Values of script Variables

While you are in the debugger, you can also enter commands, display the value of variables, use cmdlets, and run scripts at the command line.

You can display the current value of all variables in the script that is being debugged, except for the following automatic variables:

$_
$Args
$Input
$MyInvocation
$PSBoundParameters

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.

To display the value these variables for the script that is being debugged, in the script, assign the value of the automatic variable to a new variable. Then you can display the value of the new variable.

For example,

```
$scriptArgs = $Args
$scriptArgs
```

In the example in this topic, the value of the $MyInvocation variable is reassigned as follows:

```
$scriptname = $MyInvocation.MyCommand.Path
```

The Debugger Environment

When you reach a breakpoint, you enter the debugger environment. The command prompt changes so that it begins with "[DBG]:". If you are debugging a workflow, the prompt is "[WFDBG]". You can customize the prompt.

Also, in some host applications, such as the Windows PowerShell console, (but not in Windows PowerShell Integrated Scripting Environment [ISE]), a nested prompt opens for debugging. You can detect the nested prompt by the repeating greater-than characters (ASCII 62) that appear at the command prompt.

For example, the following is the default debugging prompt in the Windows PowerShell console:

```
[DBG]: PS (get-location)>>>
```

You can find the nesting level by using the $NestedPromptLevel

automatic variable.

Additionally, an automatic variable, $PSDebugContext, is defined in the local scope. You can use the presence of the $PsDebugContext variable to determine whether you are in the debugger.

For example:

```
if ($psdebugcontext) {"Debugging"} else {"Not Debugging"}
```

You can use the value of the $PSDebugContext variable in your debugging.

```
[DBG]: PS>>> $psdebugcontext.invocationinfo

 Name   CommandLineParameters  UnboundArguments  Location
 ----   ---------------------  ----------------  --------
 =      {}                     {}                C:\ps-test\vote.ps1 (1)
```

Debugging and Scope

Breaking into the debugger does not change the scope in which you are operating, but when you reach a breakpoint in a script, you move into the script scope. The script scope is a child of the scope in which you ran the debugger.

To find the variables and aliases that are defined in the script scope, use the Scope parameter of the Get-Alias or Get-Variable cmdlets.

For example, the following command gets the variables in the local (script) scope:

```
get-variable -scope 0
```

You can abbreviate the command as:

```
gv -s 0
```

This is a useful way to see only the variables that you defined in the script and that you defined while debugging.

Debugging at the Command Line

When you set a variable breakpoint or a command breakpoint, you can set the breakpoint only in a script file. However, by default, the breakpoint is set on anything that runs in the current session.

For example, if you set a breakpoint on the $name variable, the debugger breaks on any $name variable in any script, command, function, script cmdlet or expression that you run until you disable or remove the breakpoint.

This allows you to debug your scripts in a more realistic context in which they might be affected by functions, variables, and other scripts in the session and in the user's profile.

Line breakpoints are specific to script files, so they are set only in script files.

Debugging Workflows

The Windows PowerShell 4.0 debugger can be used to debug Windows PowerShell workflows, either in the Windows PowerShell console, or in Windows PowerShell ISE. There are some limitations with using the Windows PowerShell debugger to debug workflows.

-- You can view workflow variables while you are in the debugger, but setting workflow variables from within the debugger is not supported.
-- Tab completion when stopped in the workflow debugger is not available.
-- Workflow debugging works only with synchronous running of workflows from a Windows PowerShell script. You cannot debug workflows if they are running as a job (with the –AsJob parameter).
-- Other nested debugging scenarios--such as a workflow calling another workflow, or a workflow calling a script--are not implemented.

The following example demonstrates debugging a workflow. Note that when the debugger steps into the workflow function, the debugger prompt changes to [WFDBG].

```
PS C:\> Set-PSBreakpoint -Script C:\TestWFDemo1.ps1 -Line 8
```

```
ID Script         Line Command    Variable     Action
-- ------         ---- -------    --------     ------
0 TestWFDemo1.ps1   8

PS C:\> C:\TestWFDemo1.ps1
Entering debug mode. Use h or ? for help.

Hit Line breakpoint on 'C:\TestWFDemo1.ps1:8'

At C:\TestWFDemo1.ps1:8 char:5
+    Write-Output -InputObject "Now writing output:"
+    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[WFDBG:localhost]: PS C:\>> list

   3:
   4:  workflow SampleWorkflowTest
   5:  {
   6:      param ($MyOutput)
   7:
   8:*     Write-Output -InputObject "Now writing output:"
   9:      Write-Output -Input $MyOutput
  10:
  11:      Write-Output -InputObject "Get PowerShell process:"
  12:      Get-Process -Name powershell
  13:
  14:      Write-Output -InputObject "Workflow function complete."
  15:  }
  16:
  17:  # Call workflow function
  18:  SampleWorkflowTest -MyOutput "Hello"

[WFDBG:localhost]: PS C:\>> $MyOutput
Hello
[WFDBG:localhost]: PS C:\>> stepOver
Now writing output:
At C:\TestWFDemo1.ps1:9 char:5
+    Write-Output -Input $MyOutput
+    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[WFDBG:localhost]: PS C:\>> list

   4:  workflow SampleWorkflowTest
   5:  {
   6:      param ($MyOutput)
   7:
   8:      Write-Output -InputObject "Now writing output:"
   9:*     Write-Output -Input $MyOutput
  10:
```

```
11:     Write-Output -InputObject "Get PowerShell process:"
12:     Get-Process -Name powershell
13:
14:     Write-Output -InputObject "Workflow function complete."
15: }
16:
17: # Call workflow function
18: SampleWorkflowTest -MyOutput "Hello"
19:
```

```
[WFDBG:localhost]: PS C:\>> stepOver
Hello
At C:\TestWFDemo1.ps1:11 char:5
+    Write-Output -InputObject "Get PowerShell process:"
+    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[WFDBG:localhost]: PS C:\>> stepOut
Get PowerShell process:

Handles  NPM(K)    PM(K)   WS(K) VM(M)   CPU(s)     Id ProcessName    PSComputerName
-------  ------    -----   ----- -----   ------     -- -----------    --------------
    433     35   106688   128392   726    2.67   7124 powershell    localhost
    499     44   134244   172096   787    2.79   7452 powershell    localhost
Workflow function complete.
```

Debugging Functions

When you set a breakpoint on a function that has Begin, Process, and End sections, the debugger breaks at the first line of each section.

For example:

```
function test-cmdlet
{
   begin
   {
      write-output "Begin"
   }
   process
   {
      write-output "Process"
   }
   end
   {
      write-output "End"
   }
}
```

```
C:\PS> set-psbreakpoint -command test-cmdlet

C:\PS> test-cmdlet

Begin
Entering debug mode. Use h or ? for help.

Hit Command breakpoint on 'prompt:test-cmdlet'

test-cmdlet

[DBG]: C:\PS> c
Process
Entering debug mode. Use h or ? for help.

Hit Command breakpoint on 'prompt:test-cmdlet'

test-cmdlet

[DBG]: C:\PS> c
End
Entering debug mode. Use h or ? for help.

Hit Command breakpoint on 'prompt:test-cmdlet'

test-cmdlet

[DBG]: C:\PS>
```

Debugging Remote Scripts

Starting in Windows PowerShell 5.0, you can run the Windows PowerShell debugger in a remote session, in either the console, or Windows PowerShell ISE. Enter-PSSession functionality has been updated to let you reconnect to and enter a disconnected session that is running on a remote computer, and currently running a script. If the running script hits a breakpoint, your client session automatically starts the debugger. The following is an example that shows how this works, with breakpoints set in a script at lines 6, 11, 22, and 25. Note that in the example, when the debugger starts, there are two identifying prompts: the name of the computer on which the session is running, and the DBG prompt that lets you know you are in debugging mode.

```
Enter-Pssession -Cn localhost
[localhost]: PS C:\psscripts> Set-PSBreakpoint .\ttest19.ps1 6,11,22,25

ID Script        Line     Command         Variable        Action
-- ------        ----     -------         --------        ------
```

```
0 ttest19.ps1        6
1 ttest19.ps1        11
2 ttest19.ps1        22
3 ttest19.ps1        25

 [localhost]: PS C:\psscripts> .\ttest19.ps1
Hit Line breakpoint on 'C:\psscripts\ttest19.ps1:11'

At C:\psscripts\ttest19.ps1:11 char:1
+ $winRMName = "WinRM"
+ ~
 [localhost]: [DBG]: PS C:\psscripts>> list


6:     1..5 | foreach { sleep 1; Write-Output "hello2day $_" }
7:  }
8:
9:  $count = 10
10:  $psName = "PowerShell"
11:* $winRMName = "WinRM"
12:  $myVar = 102
13:
14:  for ($i=0; $i -lt $count; $i++)
15:  {
16:      sleep 1
17:      Write-Output "Loop iteration is: $i"
18:      Write-Output "MyVar is $myVar"
19:
20:      hello2day
21:

 [localhost]: [DBG]: PS C:\psscripts>> stepover
At C:\psscripts\ttest19.ps1:12 char:1
+ $myVar = 102
+ ~
 [localhost]: [DBG]: PS C:\psscripts>> quit
 [localhost]: PS C:\psscripts> Exit-PSSession
PS C:\psscripts>
```

Examples

This test script detects the version of the operating system and displays a system-appropriate message. It includes a function, a function call, and a variable.

The following command displays the contents of the test script file:

```
c:>\PS-test>  get-content test.ps1


function psversion {
        "Windows PowerShell " + $psversiontable.psversion
        if ($psversiontable.psversion.major -lt 2) {
           "Upgrade to Windows PowerShell 2.0!"
        }
        else {
           "Have you run a background job today (start-job)?"
        }
     }

$scriptname = $MyInvocation.MyCommand.Path
psversion
"Done $scriptname."
```

To start, set a breakpoint at a point of interest in the script, such as a line, command, variable, or function.

Start by creating a line breakpoint on the first line of the Test.ps1 script in the current directory.

```
PS C:\ps-test> set-psbreakpoint -line 1 -script test.ps1
```

You can abbreviate this command as:

```
PS C:\ps-test> spb 1 -s test.ps1
```

The command returns a line-breakpoint object (System.Management.Automation.LineBreakpoint).

```
Column     : 0
Line       : 1
Action     :
Enabled    : True
HitCount   : 0
Id         : 0
Script     : C:\ps-test\test.ps1
ScriptName : C:\ps-test\test.ps1
```

Now, start the script.

```
PS C:\ps-test> .\test.ps1
```

When the script reaches the first breakpoint, the breakpoint message indicates that the debugger is active. It describes the breakpoint and previews the first line of the script, which is a function declaration. The command prompt also changes to indicate that the debugger has control.

The preview line includes the script name and the line number of the previewed command.

```
Entering debug mode. Use h or ? for help.

Hit Line breakpoint on 'C:\ps-test\test.ps1:1'

test.ps1:1   function psversion {
DBG>
```

Use the Step command (s) to execute the first statement in the script and to preview the next statement. The next statement uses the $MyInvocation automatic variable to set the value of the $ScriptName variable to the path and file name of the script file.

```
DBG> s
test.ps1:11  $scriptname = $MyInvocation.MyCommand.Path
```

At this point, the $ScriptName variable is not populated, but you can verify the value of the variable by displaying its value. In this case, the value is $null.

```
DBG> $scriptname
DBG>
```

Use another Step command (s) to execute the current statement and to preview the next statement in the script. The next statement calls the

PsVersion function.

```
DBG> s
test.ps1:12  psversion
```

At this point, the $ScriptName variable is populated, but you verify the value of the variable by displaying its value. In this case, the value is set to the script path.

```
DBG> $scriptname
C:\ps-test\test.ps1
```

Use another Step command to execute the function call. Press ENTER, or type "s" for Step.

```
DBG> s
test.ps1:2       "Windows PowerShell " + $psversiontable.psversion
```

The debug message includes a preview of the statement in the function. To execute this statement and to preview the next statement in the function, you can use a Step command. But, in this case, use a Step-Out command (o). It completes the execution of the function (unless it reaches a breakpoint) and steps to the next statement in the script.

```
DBG> o
Windows PowerShell 2.0
Have you run a background job today (start-job)?
test.ps1:13  "Done $scriptname"
```

Because we are on the last statement in the script, the Step, Step-Out, and Continue commands have the same effect. In this case, use Step-Out (o).

```
Done C:\ps-test\test.ps1
PS C:\ps-test>
```

The Step-Out command executes the last command. The standard command prompt indicates that the debugger has exited and returned control to the

command processor.

Now, run the debugger again. First, to delete the current breakpoint, use the Get-PsBreakpoint and Remove-PsBreakpoint cmdlets. (If you think you might reuse the breakpoint, use the Disable-PsBreakpoint cmdlet instead of Remove-PsBreakpoint.)

```
PS C:\ps-test> Get-PsBreakpoint | Remove-PSBreakpoint
```

You can abbreviate this command as:

```
PS C:\ps-test> gbp | rbp
```

Or, run the command by writing a function, such as the following function:

```
function delbr { gbp | rbp }
```

Now, create a breakpoint on the $scriptname variable.

```
PS C:\ps-test> set-psbreakpoint -variable scriptname -script test.ps1
```

You can abbreviate the command as:

```
PS C:\ps-test> sbp -v scriptname -s test.ps1
```

Now, start the script. The script reaches the variable breakpoint. The default mode is Write, so execution stops just before the statement that changes the value of the variable.

```
PS C:\ps-test> .\test.ps1
Hit Variable breakpoint on 'C:\ps-test\test.ps1:$scriptname'
      (Write access)

test.ps1:11  $scriptname = $MyInvocation.mycommand.path
DBG>
```

Display the current value of the $scriptname variable, which is $null.

```
DBG> $scriptname
DBG>
```

Use a Step command (s) to execute the statement that populates the variable. Then, display the new value of the $scriptname variable.

```
DBG> $scriptname
C:\ps-test\test.ps1
```

Use a Step command (s) to preview the next statement in the script.

```
DBG> s
test.ps1:12  psversion
```

The next statement is a call to the PsVersion function. To skip the function but still execute it, use a Step-Over command (v). If you are already in the function when you use Step-Over, it is not effective. The function call is displayed, but it is not executed.

```
DBG> v
Windows PowerShell 2.0
Have you run a background job today (start-job)?
test.ps1:13  "Done $scriptname"
```

The Step-Over command executes the function, and it previews the next statement in the script, which prints the final line.

Use a Stop command (t) to exit the debugger. The command prompt reverts to the standard command prompt.

```
C:\ps-test>
```

To delete the breakpoints, use the Get-PsBreakpoint and Remove-PsBreakpoint cmdlets.

```
PS C:\ps-test> Get-PsBreakpoint | Remove-PSBreakpoint
```

Create a new command breakpoint on the PsVersion function.

```
PS C:\ps-test> Set-PsBreakpoint -command psversion -script test.ps1
```

You can abbreviate this command to:

```
PS C:\ps-test> sbp -c psversion -s test.ps1
```

Now, run the script.

```
PS C:\ps-test> .\test.ps1
Hit Command breakpoint on 'C:\ps-test\test.ps1:psversion'

test.ps1:12  psversion
DBG>
```

The script reaches the breakpoint at the function call. At this point, the function has not yet been called. This gives you the opportunity to use the Action parameter of Set-PsBreakpoint to set conditions for the execution of the breakpoint or to perform preparatory or diagnostic tasks, such as starting a log or invoking a diagnostic or security script.

To set an action, use a Continue command (c) to exit the script, and a Remove-PsBreakpoint command to delete the current breakpoint. (Breakpoints are read-only, so you cannot add an action to the current breakpoint.)

```
DBG> c
Windows PowerShell 2.0
Have you run a background job today (start-job)?
Done C:\ps-test\test.ps1
```

```
PS C:\ps-test> get-psbreakpoint | remove-psbreakpoint
PS C:\ps-test>
```

Now, create a new command breakpoint with an action. The following command sets a command breakpoint with an action that logs the value of the $scriptname variable when the function is called. Because the Break keyword is not used in the action, execution does not stop. (The backtick (`) is the line-continuation character.)

```
PS C:\ps-test> set-psbreakpoint -command psversion -script test.ps1  `
-action { add-content "The value of `$scriptname is $scriptname." `
-path action.log}
```

You can also add actions that set conditions for the breakpoint. In the following command, the command breakpoint is executed only if the execution policy is set to RemoteSigned, the most restrictive policy that still permits you to run scripts. (The backtick (`) is the continuation character.)

```
PS C:\ps-test> set-psbreakpoint -script test.ps1 -command psversion `
-action { if ((get-executionpolicy) -eq "RemoteSigned") { break }}
```

The Break keyword in the action directs the debugger to execute the breakpoint. You can also use the Continue keyword to direct the debugger to execute without breaking. Because the default keyword is Continue, you must specify Break to stop execution.

Now, run the script.

```
PS C:\ps-test> .\test.ps1
Hit Command breakpoint on 'C:\ps-test\test.ps1:psversion'

test.ps1:12  psversion
```

Because the execution policy is set to RemoteSigned, execution stops at the function call.

At this point, you might want to check the call stack. Use the Get-PsCallStack cmdlet or the Get-PsCallStack debugger command (k).

The following command gets the current call stack.

```
DBG> k
2: prompt
1: .\test.ps1: $args=[]
0: prompt: $args=[]
```

This example demonstrates just a few of the many ways to use the Windows PowerShell debugger.

For more information about the debugger cmdlets, type the following command:

```
help <cmdlet-name> -full
```

For example, type:

```
help set-psbreakpoint -full
```

Other Debugging Features in Windows PowerShell

In addition to the Windows PowerShell debugger, Windows PowerShell includes several other features that you can use to debug scripts and functions.

-- Windows PowerShell Integrated Scripting Environment (ISE) includes an interactive graphical debugger. For more information, start Windows PowerShell ISE and press F1.

-- The Set-PSDebug cmdlet offers very basic script debugging features, including stepping and tracing.

-- Use the Set-StrictMode cmdlet to detect references to uninitialized variables, to references to non-existent properties of an object, and to function syntax that is not valid.

-- Add diagnostic statements to a script, such as statements that display the value of variables, statements that read input from the command line, or statements that report the current instruction. Use the cmdlets that contain the Write verb for this task, such as Write-Host, Write-Debug, Write-Warning, and

Write-Verbose.

SEE ALSO
Disable-PsBreakpoint
Enable-PsBreakpoint
Get-PsBreakpoint
Get-PsCallStack
Remove-PsBreakpoint
Set-PsBreakpoint
Set-PsDebug
Set-Strictmode
Write-Debug
Write-Verbose

about_Desired_State_Configuration

SHORT DESCRIPTION
Provides a brief introduction to the Windows
PowerShell Desired State Configuration (DSC) feature.

LONG DESCRIPTION
DSC is a management platform in Windows PowerShell that enables deploying and managing configuration data for software services, and managing the environment in which these services run.

DSC provides a set of Windows PowerShell language extensions, new cmdlets, and resources that you can use to declaratively specify how you want the state of your software environment to be configured. It also provides a means to maintain and manage existing configurations.

DSC is introduced in Windows PowerShell 4.0.

For detailed information about DSC, see "Windows PowerShell Desired State Configuration Overview" in the TechNet Library at http://go.microsoft.com/fwlink/?LinkId=311940.

DEVELOPING DSC RESOURCES WITH CLASSES
Starting in Windows PowerShell 5.0, you can develop DSC resources by using classes. For more information, see about_Classes, and "Writing a custom DSC resource with PowerShell classes" on Microsoft TechNet (http://technet.microsoft.com/library/dn948461.aspx).

USING DSC
To use DSC to configure your environment, first define a Windows PowerShell script block using the Configuration keyword, followed by an identifier, which is in turn followed by the pair of curly braces delimiting

the block. Inside the configuration block you can define node blocks that specify the desired configuration state for each node (computer) in the environment. A node block starts with the Node keyword, followed by the name of the target computer, which can be a variable. After the computer name, come the curly braces that delimit the node block. Inside the node block, you can define resource blocks to configure specific resources. A resource block starts with the type name of the resource, followed by the identifier you want to specify for that block, followed by the curly braces that delimit the block, as shown in the following example.

```
Configuration MyWebConfig
{
  # Parameters are optional
  param ($MachineName, $WebsiteFilePath)

  # A Configuration block can have one or more Node blocks
  Node $MachineName
  {
    # Next, specify one or more resource blocks
    # WindowsFeature is one of the resources you can use in a Node block
    # This example ensures the Web Server (IIS) role is installed
    WindowsFeature IIS
    {
      # To ensure that the role is not installed, set Ensure to "Absent"
       Ensure = "Present"
       Name = "Web-Server" # Use the Name property from Get-WindowsFeature
    }

    # You can use the File resource to create files and folders
    # "WebDirectory" is the name you want to use to refer to this instance
    File WebDirectory
    {
      Ensure = "Present"  # You can also set Ensure to "Absent“
      Type = "Directory“ # Default is “File”
      Recurse = $true
      SourcePath = $WebsiteFilePath
      DestinationPath = "C:\inetpub\wwwroot"

      # Ensure that the IIS block is successfully run first before
      # configuring this resource
      Requires = "[WindowsFeature]IIS"  # Use Requires for dependencies
    }
  }
}
```

To create a configuration, invoke the Configuration block the same way you would invoke a Windows PowerShell function, passing in any expected parameters you may have defined (two in the example above). For example, in this case:

```
MyWebConfig -MachineName "TestMachine" –WebsiteFilePath "\\filesrv\WebFiles" `
    -OutputPath "C:\Windows\system32\temp" # OutputPath is optional
```

This generates a MOF file per node at the path you specify. These MOF files specify the desired configuration for each node. Next, use the following cmdlet to parse the configuration MOF files, send each node its corresponding configuration, and enact those configurations. Note that you do not need to create a separate MOF file for class-based DSC resources.

```
Start-DscConfiguration –Verbose -Wait -Path "C:\Windows\system32\temp"
```

USING DSC TO MAINTAIN CONFIGURATION STATE

With DSC, configuration is idempotent. This means that if you use DSC to enact the same configuration more than once, the resulting configuration state will always be the same. Because of this, if you suspect that any nodes in your environment may have drifted from the desired state of configuration, you can enact the same DSC configuration again to bring them back to the desired state. You do not need to modify the configuration script to address only those resources whose state has drifted from the desired state.

The following example shows how you can verify whether the actual state of configuration on a given node has drifted from the last DSC configuration enacted on the node. In this example we are checking the configuration of the local computer.

```
$session = New-CimSession -ComputerName "localhost"
Test-DscConfiguration -CimSession $session
```

BUILT-IN DSC RESOURCES

You can use the following built-in resources in your configuration scripts:

```
Name               Properties
----               ----------
File               {DestinationPath, Attributes, Checksum, Contents...}
Archive            {Destination, Path, Checksum, Credential...}
Environment        {Name, DependsOn, Ensure, Path...}
Group              {GroupName, Credential, DependsOn, Description...}
Log                {Message, DependsOn, PsDscRunAsCredential}
Package            {Name, Path, ProductId, Arguments...}
Registry           {Key, ValueName, DependsOn, Ensure...}
Script             {GetScript, SetScript, TestScript, Credential...}
Service            {Name, BuiltInAccount, Credential, Dependencies...}
User               {UserName, DependsOn, Description, Disabled...}
WaitForAll         {NodeName, ResourceName, DependsOn, PsDscRunAsCredential...}
WaitForAny         {NodeName, ResourceName, DependsOn, PsDscRunAsCredential...}
WaitForSome        {NodeCount, NodeName, ResourceName, DependsOn...}
WindowsFeature     {Name, Credential, DependsOn, Ensure...}
WindowsOptionalFeature {Name, DependsOn, Ensure, LogLevel...}
```

WindowsProcess {Arguments, Path, Credential, DependsOn...}

To get a list of available DSC resources on your system, run the Get-DscResource cmdlet.

The example in this topic demonstrates how to use the File and WindowsFeature resources. To see all properties that you can use with a resource, insert the cursor in the resource keyword (for example, File) within your configuration script in Windows PowerShell ISE, hold down CTRL, and then press SPACEBAR.

FIND MORE RESOURCES

You can download, install, and learn about many other available DSC resources that have been created by the PowerShell and DSC user community, and by Microsoft. Visit the PowerShell Gallery (https://www.powershellgallery.com/) to browse and learn about available DSC resources.

SEE ALSO

"Windows PowerShell Desired State Configuration Overview" (http://go.microsoft.com/fwlink/?LinkId=311940)
"Built-In Windows PowerShell Desired State Configuration Resources" (http://technet.microsoft.com/library/dn249921.aspx)
"Build Custom Windows PowerShell Desired State Configuration Resources" (http://technet.microsoft.com/library/dn249927.aspx)

about_Do

SHORT DESCRIPTION

Runs a statement list one or more times, subject to a While or Until condition.

LONG DESCRIPTION

The Do keyword works with the While keyword or the Until keyword to run the statements in a script block, subject to a condition. Unlike the related While loop, the script block in a Do loop always runs at least

once.

A Do-While loop is a variety of the While loop. In a Do-While loop, the condition is evaluated after the script block has run. As in a While loop, the script block is repeated as long as the condition evaluates to true.

Like a Do-While loop, a Do-Until loop always runs at least once before the condition is evaluated. However, the script block runs only while the condition is false.

The Continue and Break flow control keywords can be used in a Do-While loop or in a Do-Until loop.

Syntax

The following shows the syntax of the Do-While statement:

```
do {<statement list>} while (<condition>)
```

The following shows the syntax of the Do-Until statement:

```
do {<statement list>} until (<condition>)
```

The statment list contains one or more statements that run each time the loop is entered or repeated.

The condition portion of the statement resolves to true or false.

Example

The following example of a Do statement counts the items in an array until it reaches an item with a value of 0.

```
C:\PS> $x = 1,2,78,0
C:\PS> do { $count++; $a++; } while ($x[$a] -ne 0)
C:\PS> $count
3
```

The following example uses the Until keyword. Notice that the not equal to operator (-ne) is replaced by the equal to operator (-eq).

```
C:\PS> $x = 1,2,78,0
C:\PS> do { $count++; $a++; } until ($x[$a] -eq 0)
C:\PS> $count
3
```

The following example writes all the values of an array, skipping any value that is less than zero.

```
do
{
   if ($x[$a] -lt 0) { continue }
   Write-Host $x[$a]
}
while (++$a -lt 10)
```

SEE ALSO

about_While
about_Operators
about_Assignment_Operators
about_Comparison_Operators
about_Break
about_Continue

about_Environment_Variables

SHORT DESCRIPTION

Describes how to access Windows environment variables in Windows PowerShell.

LONG DESCRIPTION

Environment variables store information about the operating system environment. This information includes details such as the operating system path, the number of processors used by the operating system, and the location of temporary folders.

The environment variables store data that is used by the operating system and other programs. For example, the WINDIR environment variable contains the location of the Windows installation directory. Programs can query the value of this variable to determine where Windows operating system files are located.

Windows PowerShell lets you view and change Windows environment variables, including the variables set in the registry, and those set for a particular session. The Windows PowerShell environment provider simplifies this process by making it easy to view and change the environment variables.

Unlike other types of variables in Windows PowerShell, environment variables and their values are inherited by child sessions, such as local background jobs and the sessions in which module members run. This makes environment variables well suited to storing values that are needed in both parent and child sessions.

Windows PowerShell Environment Provider

The Windows PowerShell environment provider lets you access Windows environment variables in Windows PowerShell in a Windows PowerShell drive (the Env: drive). This drive looks much like a file system drive. To go to the Env: drive, type:

```
Set-Location Env:
```

Then, to display the contents of the Env: drive, type:

```
Get-ChildItem
```

You can view the environment variables in the Env: drive from any other Windows PowerShell drive, and you can go into the Env: drive to view and change the environment variables.

Environment Variable Objects

In Windows PowerShell, each environment variable is represented by an object that is an instance of the System.Collections.DictionaryEntry class.

In each DictionaryEntry object, the name of the environment variable is the dictionary key. The value of the variable is the dictionary value.

To display an environment variable in Windows PowerShell, get an object that represents the variable, and then display the values of the object properties. When you change an environment variable in Windows PowerShell, use the methods that are associated with the DictionaryEntry object.

To display the properties and methods of the object that represents an environment variable in Windows PowerShell, use the Get-Member cmdlet. For example, to display the methods and properties of all the objects in the Env: drive, type:

```
Get-Item -Path Env:* | Get-Member
```

Displaying Environment Variables

You can use the cmdlets that contain the Item noun (the Item cmdlets) to display and change the values of environment variables. Because environment variables do not have child items, the output of Get-Item and Get-ChildItem is the same.

When you refer to an environment variable, type the Env: drive name followed by the name of the variable. For example, to display the value of the COMPUTERNAME environment variable, type:

```
Get-Childitem Env:Computername
```

To display the values of all the environment variables, type:

```
Get-ChildItem Env:
```

By default, Windows PowerShell displays the environment variables in the order in which it retrieves them. To sort the list of environment variables by variable name, pipe the output of a Get-ChildItem command to

the Sort-Object cmdlet. For example, from any Windows PowerShell drive, type:

```
Get-ChildItem Env: | Sort Name
```

You can also go into the Env: drive by using the Set-Location cmdlet:

```
Set-Location Env:
```

When you are in the Env: drive, you can omit the Env: drive name from the path. For example, to display all the environment variables, type:

```
Get-ChildItem
```

To display the value of the COMPUTERNAME variable from within the Env: drive, type:

```
Get-ChildItem ComputerName
```

You can also display and change the values of environment variables without using a cmdlet by using the expression parser in Windows PowerShell. To display the value of an environment variable, use the following syntax:

```
$Env:<variable-name>
```

For example, to display the value of the WINDIR environment variable, type the following command at the Windows PowerShell command prompt:

```
$Env:windir
```

In this syntax, the dollar sign ($) indicates a variable, and the drive name indicates an environment variable.

Changing Environment Variables

To make a persistent change to an environment variable, use System in

Control Panel (Advanced tab or the Advanced System Settings item) to store the change in the registry.

When you change environment variables in Windows PowerShell, the change affects only the current session. This behavior resembles the behavior of the Set command in Windows-based environments and the Setenv command in UNIX-based environments.

You must also have permission to change the values of the variables. If you try to change a value without sufficient permission, the command fails, and Windows PowerShell displays an error.

You can change the values of variables without using a cmdlet by using the following syntax:

```
$Env:<variable-name> = "<new-value>"
```

For example, to append ";c:\temp" to the value of the Path environment variable, use the following syntax:

```
$Env:path = $env:path + ";c:\temp"
```

You can also use the Item cmdlets, such as Set-Item, Remove-Item, and Copy-Item to change the values of environment variables. For example, to use the Set-Item cmdlet to append ";c:\temp" to the value of the Path environment variable, use the following syntax:

```
Set-Item -Path Env:Path -Value ($Env:Path + ";C:\Temp")
```

In this command, the value is enclosed in parentheses so that it is interpreted as a unit.

Saving Changes to Environment Variables

To create or change the value of an environment variable in every Windows PowerShell session, add the change to your Windows PowerShell profile.

For example, to add the C:\Temp directory to the Path environment variable in every Windows PowerShell session, add the following command to your Windows PowerShell profile.

$Env:Path = $Env:Path + ";C:\Temp"

To add the command to an existing profile, such as the CurrentUser,AllHosts profile, type:

Add-Content -Path $Profile.CurrentUserAllHosts -Value '$Env:Path = $Env:Path + ";C:\Temp"'

Environment Variables That Store Preferences

Windows PowerShell features can use environment variables to store user preferences. These variables work like preference variables, but they are inherited by child sessions of the sessions in which they are created. For more information about preference variables, see about_preference_variables.

The environment variables that store preferences include:

PSExecutionPolicyPreference

Stores the execution policy set for the current session. This environment variable exists only when you set an execution policy for a single session. You can do this in two different ways.

-- Use PowerShell.exe to start a session at the command line and use its ExecutionPolicy parameter to set the execution policy for the session.

-- Use the Set-ExecutionPolicy cmdlet. Use the Scope parameter with a value of "Process".

For more information, see about_Execution_Policies.

PSModulePath

Stores the paths to the default module directories. Windows PowerShell looks for modules in the specified directories when you do not specify a full path to a module.

The default value of $Env:PSModulePath is:

$home\Documents\WindowsPowerShell\Modules; $pshome\Modules

Windows PowerShell sets the value of "$pshome\Modules" in the registry. It sets the value of "$home\Documents\WindowsPowerShell\Modules" each time you start Windows PowerShell.

In addition, setup programs that install modules in other directories, such as the Program Files directory, can append their locations to the value of PSModulePath.

To change the default module directories for the current session, use the following command format to change the value of the PSModulePath environment variable.

For example, to add the "C:\Program Files\Fabrikam\Modules" directory to the value of the PSModulePath environment variable, type:

```
$Env:PSModulePath = $Env:PSModulePath + ";C:\Program Files\Fabrikam\Modules"
```

The semi-colon (;) in the command separates the new path from the path that precedes it in the list.

To change the value of PSModulePath in every session, add the previous command to your Windows PowerShell profile or use the SetEnvironmentVariable method of the Environment class.

The following command uses the GetEnvironmentVariable method to get the machine setting of PSModulePath and the SetEnvironmentVariable method to add the C:\Program Files\Fabrikam\Modules path to the value.

```
$path = [System.Environment]::GetEnvironmentVariable("PSModulePath", "Machine")
[System.Environment]::SetEnvironmentVariable("PSModulePath", $path + `
  ";C:\Program Files\Fabrikam\Modules", "Machine")
```

To add a path to the user setting, change the target value to User.

```
$path = [System.Environment]::GetEnvironmentVariable("PSModulePath", "User")
[System.Environment]::SetEnvironmentVariable("PSModulePath", $path + `
  ";$home\Documents\Fabrikam\Modules", "User")
```

For more information about the methods of the System.Environment class, see "Environment Methods" in MSDN at http://go.microsoft.com/fwlink/?LinkId=242783.

You can add also add a command that changes the value to your profile or use System in Control Panel to change the value of the PSModulePath environment variable in the registry.

For more information, see about_Modules.

SEE ALSO

- Environment (provider)
- about_Modules

about_Escape_Characters

SHORT DESCRIPTION

Introduces the escape character in Windows PowerShell and explains its effect.

LONG DESCRIPTION

Escape characters are used to assign a special interpretation to the characters that follow it.

In Windows PowerShell, the escape character is the backtick (`), also called the grave accent (ASCII 96). The escape character can be used to indicate a literal, to indicate line continuation, and to indicate special characters.

In a call to another program, instead of using escape characters to prevent Windows PowerShell from misinterpreting program arguments, you can use the stop-parsing symbol (--%). The stop-parsing symbol is introduced in Windows PowerShell 3.0.

ESCAPING A VARIABLE

When an escape character precedes a variable, it prevents a value from being substituted for the variable.

For example:

```
PS C:\>$a = 5
PS C:\>"The value is stored in $a."
The value is stored in 5.

PS C:\>$a = 5
PS C:\>"The value is stored in `$a."
The value is stored in $a.
```

ESCAPING QUOTATION MARKS

When an escape character precedes a
double quotation mark, Windows PowerShell interprets the double quotation mark as a character, not as a string delimiter.

```
PS C:\> "Use quotation marks (") to indicate a string."
Unexpected token ')' in expression or statement.
At line:1 char:25
+ "Use quotation marks (") <<<<  to indicate a string."

PS C:\> "Use quotation marks (`") to indicate a string."
Use quotation marks (") to indicate a string.
```

USING LINE CONTINUATION

The escape character tells Windows PowerShell that the command continues on the next line.

For example:

```
PS C:\> Get-Process `
>> PowerShell

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    340       8    34556      31864   149     0.98   2036 PowerShell
```

USING SPECIAL CHARACTERS

When used within quotation marks, the escape character indicates a special character that provides instructions to the command parser.

The following special characters are recognized by Windows PowerShell:

- `0 Null
- `a Alert
- `b Backspace
- `f Form feed
- `n New line
- `r Carriage return
- `t Horizontal tab
- `v Vertical tab

For example:

```
PS C:\> "12345678123456781`nCol1`tColumn2`tCol3"
12345678123456781
```

Col1 Column2 Col3

For more information, type:
Get-Help about_Special_Characters

STOP-PARSING SYMBOL

When calling other programs, you can use the stop-parsing symbol (--%) to prevent Windows PowerShell from generating errors or misinterpreting program arguments. The stop-parsing symbol is an alternative to using escape characters in program calls. It is introduced in Windows PowerShell 3.0.

For example, the following command uses the stop-parsing symbol in an Icacls command:

```
icacls X:\VMS --% /grant Dom\HVAdmin:(CI)(OI)F
```

For more information about the stop-parsing symbol, see about_Parsing.

SEE ALSO

about_Quoting_Rules

about_EventLogs

SHORT DESCRIPTION

Windows PowerShell creates a Windows event log that is named "Windows PowerShell" to record Windows PowerShell events. You can view this log in Event Viewer or by using cmdlets that get events, such as the Get-EventLog cmdlet. By default, Windows PowerShell engine and provider events are recorded in the event log, but you can use the event log preference variables to customize the event log. For example, you can add events about Windows PowerShell commands.

LONG DESCRIPTION

The Windows PowerShell event log records details of Windows PowerShell operations, such as starting and stopping the program engine and starting

and stopping the Windows PowerShell providers. You can also log details about Windows PowerShell commands.

The Windows PowerShell event log is in the Application and Services Logs group. The Windows PowerShell log is a classic event log that does not use the Windows Eventing technology. To view the log, use the cmdlets designed for classic event logs, such as Get-EventLog.

Viewing the Windows PowerShell Event Log

You can view the Windows PowerShell event log in Event Viewer or by using the Get-EventLog and Get-WmiObject cmdlets. To view the contents of the Windows PowerShell log, type:

```
Get-EventLog -LogName "Windows PowerShell"
```

To examine the events and their properties, use the Sort-Object cmdlet, the Group-Object cmdlet, and the cmdlets that contain the Format verb (the Format cmdlets).

For example, to view the events in the log grouped by the event ID, type:

```
Get-EventLog "Windows PowerShell" | Format-Table -GroupBy EventID
```

Or, type:

```
Get-EventLog "Windows PowerShell" | Sort-Object EventID `
    | Group-Object EventID
```

To view all the classic event logs, type:

```
Get-EventLog -List
```

You can also use the Get-WmiObject cmdlet to use the event-related Windows Management Instumentation (WMI) classes to examine the event log. For example, to view all the properties of the event log file, type:

```
Get-WmiObject Win32_NTEventLogFile | where `
    {$_.LogFileName -eq "Windows PowerShell"} | Format-List -Property *
```

To find the Win32 event-related WMI classes, type:

```
Get-WmiObject -List | where {$_.Name -like "win32*event*"}
```

For more information, type "Get-Help Get-EventLog" and
"Get-Help Get-WmiObject".

Selecting Events for the Windows PowerShell Event Log

You can use the event log preference variables to determine which events are recorded in the Windows PowerShell event log.

There are six event log preference variables; two variables for each of the three logging components: the engine (the Windows PowerShell program), the providers, and the commands. The LifeCycleEvent variables log normal starting and stopping events. The Health variables log error events.

The following table lists the event log preference variables.

Variable	Description
$LogEngineLifeCycleEvent	Logs starting and stopping of Windows PowerShell.
$LogEngineHealthEvent	Logs Windows PowerShell program errors.
$LogProviderLifeCycleEvent	Logs starting and stopping of Windows PowerShell providers.
$LogProviderHealthEvent	Logs Windows PowerShell provider errors.
$LogCommandLifeCycleEvent	Logs starting and completion of commands.
$LogCommandHealthEvent	Logs command errors.

(For information about Windows PowerShell providers,
type: "Get-Help about_providers".)

By default, only the following event types are enabled:

$LogEngineLifeCycleEvent
$LogEngineHealthEvent
$LogProviderLifeCycleEvent
$LogProviderHealthEvent

To enable an event type, set the preference variable for that event type to $true. For example, to enable command life-cycle events, type:

$LogCommandLifeCycleEvent

Or, type:

$LogCommandLifeCycleEvent = $true

To disable an event type, set the preference variable for that event type to $false. For example, to disable command life-cycle events, type:

$LogProviderLifeCycleEvent = $false

You can disable any event, except for the events that indicate that the Windows PowerShell engine and the core providers are started. These events are generated before the Windows PowerShell profiles are run and before the host program is ready to accept commands.

The variable settings apply only for the current Windows PowerShell session. To apply them to all Windows PowerShell sessions, add them to your Windows PowerShell profile.

Logging Module Events

Beginning in Windows PowerShell 3.0, you can record execution events for the cmdlets and functions in Windows PowerShell modules and snap-ins by setting the LogPipelineExecutionDetails property of modules and snap-ins to TRUE. In Windows PowerShell 2.0, this feature is available only for snap-ins.

When the LogPipelineExecutionDetails property value is TRUE ($True), Windows PowerShell writes cmdlet and function execution events in the session to the Windows PowerShell log in Event Viewer. The setting is effective only in the current session.

To enable logging of execution events of cmdlets and functions in a module, use the following command sequence.

```
Import-Module <ModuleName>
$m = Get-Module <ModuleName>
$m.LogPipelineExecutionDetails = $True
```

To enable logging of execution events of cmdlets in a snap-in, use the following command sequence.

```
$m = Get-PSSnapin <SnapInName> [-Registered]
$m.LogPipelineExecutionDetails = $True
```

To disable logging, use the same command sequence to set the property value to FALSE ($False).

You can also use the "Turn on Module Logging" Group Policy setting to enable and disable module and snap-in logging. The policy value includes a list of module and snap-in names. Wildcards are supported.

When "Turn on Module Logging" is set for a module, the value of the LogPipelineExecutionDetails property of the module is TRUE in all sessions and it cannot be changed.

The Turn On Module Logging group policy setting is located in the following Group Policy paths:

```
Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell
User Configuration\Administrative Templates\Windows Components\Windows PowerShell
```

The User Configuration policy takes precedence over the Computer Configuration policy, and both policies take preference over the value of the LogPipelineExecutionDetails property of modules and snap-ins.

For more information about this Group Policy setting, see about_Group_Policy_Settings (http://go.microsoft.com/fwlink/?LinkID=251696).

Security and Auditing

The Windows PowerShell event log is designed to indicate activity and to provide operational details for troubleshooting.

However, like most Windows-based application event logs, the Windows PowerShell event log is not designed to be secure. It should not be used to audit security or to record confidential or proprietary information.

Event logs are designed to be read and understood by users. Users can read from and write to the log. A malicious user could read an event log on a local or remote computer, record false data, and then prevent the logging of their activities.

NOTES

Authors of module authors can add logging features to
their modules. For more information, see "Writing a Windows PowerShell Module" in
MSDN at http://go.microsoft.com/fwlink/?LinkID=144916.

SEE ALSO

Get-EventLog
Get-WmiObject
about_Group_Policy_Settings
about_Preference_Variables

about_Execution_Policies

SHORT DESCRIPTION

Describes the Windows PowerShell execution policies and explains
how to manage them.

LONG DESCRIPTION

Windows PowerShell execution policies let you determine the
conditions under which Windows PowerShell loads configuration files
and runs scripts.

You can set an execution policy for the local computer, for the current
user, or for a particular session. You can also use a Group Policy
setting to set execution policy for computers and users.

Execution policies for the local computer and current user are stored
in the registry. You do not need to set execution policies in your
Windows PowerShell profile. The execution policy for a particular session
is stored only in memory and is lost when the session is closed.

The execution policy is not a security system that restricts user actions.
For example, users can easily circumvent a policy by typing the script

contents at the command line when they cannot run a script. Instead, the execution policy helps users to set basic rules and prevents them from violating them unintentionally.

WINDOWS POWERSHELL EXECUTION POLICIES

The Windows PowerShell execution policies are as follows:

"Restricted" is the default policy.

Restricted

- Default execution policy in Windows 8, Windows Server 2012, and Windows 8.1.

- Permits individual commands, but will not run scripts.

- Prevents running of all script files, including formatting and configuration files (.ps1xml), module script files (.psm1), and Windows PowerShell profiles (.ps1).

AllSigned

- Scripts can run.

- Requires that all scripts and configuration files be signed by a trusted publisher, including scripts that you write on the local computer.

- Prompts you before running scripts from publishers that you have not yet classified as trusted or untrusted.

- Risks running signed, but malicious, scripts.

RemoteSigned

- Scripts can run. This is the default execution policy in Windows Server 2012 R2.

- Requires a digital signature from a trusted publisher on scripts and configuration files that are downloaded from the Internet (including e-mail and instant messaging programs).

- Does not require digital signatures on scripts that

you have written on the local computer (not downloaded from the Internet).

- Runs scripts that are downloaded from the Internet and not signed, if the scripts are unblocked, such as by using the Unblock-File cmdlet.

- Risks running unsigned scripts from sources other than the Internet and signed, but malicious, scripts.

Unrestricted

- Unsigned scripts can run. (This risks running malicious scripts.)

- Warns the user before running srcipts and configuration files that are downloaded from the Internet.

Bypass

- Nothing is blocked and there are no warnings or prompts.

- This execution policy is designed for configurations in which a Windows PowerShell script is built in to a a larger application or for configurations in which Windows PowerShell is the foundation for a program that has its own security model.

Undefined

- There is no execution policy set in the current scope.

- If the execution policy in all scopes is Undefined, the effective execution policy is Restricted, which is the default execution policy.

Note: On systems that do not distinguish Universal Naming Convention (UNC) paths from Internet paths, scripts that are identified by a UNC path might not be permitted to run with the RemoteSigned execution policy.

EXECUTION POLICY SCOPE

You can set an execution policy that is effective only in a particular scope.

The valid values for Scope are Process, CurrentUser, and LocalMachine. LocalMachine is the default when setting an execution policy.

The Scope values are listed in precedence order.

- Process
  The execution policy affects only the current session (the current Windows PowerShell process).

  The execution policy is stored in the $env:PSExecutionPolicyPreference environment variable, not in the registry, and it is deleted when the session is closed. You cannot change the policy by editing the variable value.

- CurrentUser
  The execution policy affects only the current user. It is stored in the HKEY_CURRENT_USER registry subkey.

- LocalMachine
  The execution policy affects all users on the current computer. It is stored in the HKEY_LOCAL_MACHINE registry subkey.

The policy that takes precedence is effective in the current session, even if a more restrictive policy was set at a lower level of precedence.

For more information, see Set-ExecutionPolicy.

GET YOUR EXECUTION POLICY

To get the Windows PowerShell execution policy that is in effect in the current session, use the Get-ExecutionPolicy cmdlet.

The following command gets the current execution policy:

```
Get-ExecutionPolicy
```

To get all of the execution policies that affect the current session and displays them in precedence order, type:

```
Get-ExecutionPolicy -List
```

The result will look similar to the following sample output:

```
        Scope    ExecutionPolicy
        -----    ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser      RemoteSigned
 LocalMachine        AllSigned
```

In this case, the effective execution policy is RemoteSigned because the execution policy for the current user takes precedence over the execution policy set for the local computer.

To get the execution policy set for a particular scope, use the Scope parameter of Get-ExecutionPolicy.

For example, the following command gets the execution policy for the current user scope.

```
Get-ExecutionPolicy -Scope CurrentUser
```

CHANGE YOUR EXECUTION POLICY

To change the Windows PowerShell execution policy on your computer, use the Set-ExecutionPolicy cmdlet.

The change is effective immediately; you do not need to restart Windows PowerShell.

If you set the execution policy for the local computer (the default) or the current user, the change is saved in the registry and remains effective until you change it again.

If you set the execution policy for the current process, it is not saved in the registry. It is retained until the current process and any child processes are closed.

Note: In Windows Vista and later versions of Windows, to run commands that change the execution policy for the local computer (the default), start Windows PowerShell with the "Run as administrator" option.

To change your execution policy, type:

Set-ExecutionPolicy -ExecutionPolicy <PolicyName>

For example:

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned

To set the execution policy in a particular scope, type:

Set-ExecutionPolicy -ExecutionPolicy <PolicyName> -Scope <scope>

For example:

Set-ExecutionPolicy RemoteSigned -Scope CurrentUser

A command to change an execution policy can succeed but still not change the effective execution policy.

For example, a command that sets the execution policy for the local computer can succeed but be overridden by the execution policy for the current user.

REMOVE YOUR EXECUTION POLICY

To remove the execution policy for a particular scope, set the value of the value of the execution policy to Undefined.

For example, to remove the execution policy for all the users of the local computer, type:

Set-ExecutionPolicy Undefined

Or, type:

Set-ExecutionPolicy Undefined -scope LocalMachine

If no execution policy is set in any scope, the effective execution policy is Restricted, which is the default.

SET A DIFFERENT EXECUTION POLICY FOR ONE SESSION

You can use the ExecutionPolicy parameter of PowerShell.exe to set an execution policy for a new Windows PowerShell session. The policy affects only the current session and child sessions.

To set the execution policy for a new session, start Windows PowerShell at the command line (such as Cmd.exe or Windows PowerShell), and then use the ExecutionPolicy parameter of PowerShell.exe to set the execution policy.

For example:

```
PowerShell.exe -ExecutionPolicy AllSigned
```

The execution policy that you set is not stored in the registry. Instead, it is stored in the $env:PSExecutionPolicyPreference environment variable. The variable is deleted when you close the session in which the policy is set. You cannot change the policy by editing the variable value.

During the session, the execution policy that is set for the session takes precedence over an execution policy that is set in the registry for the local computer or current user. However, it does not take precedence over the execution policy set by using a Group Policy setting (discussed below).

USE GROUP POLICY TO MANAGE EXECUTION POLICY

You can use the "Turn on Script Execution" Group Policy setting to manage the execution policy of computers in your enterprise. The Group Policy setting overrides the execution policies set in Windows PowerShell in all scopes.

The "Turn on Script Execution" policy settings are as follows:

-- If you disable "Turn on Script Execution", scripts do not run. This is equivalent to the "Restricted" execution policy.

-- If you enable "Turn on Script Execution", you can select an execution policy. The Group Policy settings are equivalent to the following execution policy settings.

Group Policy	Execution Policy

---------- ----------
Allow all scripts. Unrestricted

Allow local scripts and remote signed scripts. RemoteSigned

Allow only signed scripts. AllSigned

-- If "Turn on Script Execution" is not configured, it has no effect. The execution policy set in Windows PowerShell is effective.

The PowerShellExecutionPolicy.adm and PowerShellExecutionPolicy.admx files add the "Turn on Script Execution" policy to the Computer Configuration and User Configuration nodes in Group Policy Editor in the following paths.

For Windows XP and Windows Server 2003:
Administrative Templates\Windows Components\Windows PowerShell

For Windows Vista and later versions of Windows:
Administrative Templates\Classic Administrative Templates\
Windows Components\Windows PowerShell

Policies set in the Computer Configuration node take precedence over policies set in the User Configuration node.

The PowerShellExecutionPolicy.adm file is available on the Microsoft Download Center. For more information, see "Administrative Templates for Windows PowerShell" at http://go.microsoft.com/fwlink/?LinkId=131786.

For more information, see about_Group_Policy_Settings at http://go.microsoft.com/fwlink/?LinkID=251696.

EXECUTION POLICY PRECEDENCE

When determining the effective execution policy for a session, Windows PowerShell evaluates the execution policies in the following precedence order:

- Group Policy: Computer Configuration
- Group Policy: User Configuration
- Execution Policy: Process (or PowerShell.exe -ExecutionPolicy)
- Execution Policy: CurrentUser

- Execution Policy: LocalMachine

MANAGE SIGNED AND UNSIGNED SCRIPTS

If your Windows PowerShell execution policy is RemoteSigned, Windows PowerShell will not run unsigned scripts that are downloaded from the Internet (including e-mail and instant messaging programs).

You can sign the script or elect to run an unsigned script without changing the execution policy.

Beginning in Windows PowerShell 3.0, you can use the Stream parameter of the Get-Item cmdlet to detect files that are blocked because they were downloaded from the Internet, and you can use the Unblock-File cmdlet to unblock the scripts so that you can run them in Windows PowerShell.

For more information, see about_Signing, Get-Item, and Unblock-File.

SEE ALSO

about_Environment_Variables
about_Signing
Get-ExecutionPolicy
Set-ExecutionPolicy
Get-Item
Unblock-File
"Administrative Templates for Windows PowerShell"
(http://go.microsoft.com/fwlink/?LinkId=131786)
"PowerShell.exe Console Help"
(http://go.microsoft.com/fwlink/?LinkID=113439)

about_For

SHORT DESCRIPTION

Describes a language command you can use to run statements based on a conditional test.

LONG DESCRIPTION

The For statement (also known as a For loop) is a language construct you can use to create a loop that runs commands in a command block while a specified condition evaluates to true.

A typical use of the For loop is to iterate an array of values and to operate on a subset of these values. In most cases, if you want to iterate all the values in an array, consider using a Foreach statement.

Syntax

The following shows the For statement syntax.

```
for (<init>; <condition>; <repeat>)
{<statement list>}
```

The <init> placeholder represents one or more commands, separated by commas, that are run before the loop begins. You typically use the <init> portion of the statement to create and initialize a variable with a starting value. Note that the comma syntax doesn't work with assignment statements, such as the following example:

```
$ofs=",";$rs = "rs"; $cs = "cs"; for ($r = $rs, $c = $cs; $true;)
{ "r is '$r' and c is '$c'"; break }
```

This variable will then be the basis for the condition to be tested in the next portion of the For statement.

The <condition> placeholder represents the portion of the For statement that resolves to a true or false Boolean value. Windows PowerShell evaluates the condition each time the For loop runs. If the statement is true, the commands in the command block run, and the statement is evaluated again. If the condition is still true, the commands in the statement list run again. The loop is repeated until the condition becomes false.

The <repeat> placeholder represents one or more commands, separated by commas, that are executed each time the loop repeats. Typically, this is used to modify a variable that is tested inside the <condition> part of the statement.

The <statement list> placeholder represents a set of one or more commands that are run each time the loop is entered or repeated. The contents of the statement list are surrounded by braces.

Examples

At a minimum, a For statement requires the parenthesis surrounding the <init>, <condition>, and <repeat> part of the statement and a command surrounded by braces in the <statement list> part of the statement.

Note that the upcoming examples intentionally show code outside the For statement. In later examples, code is integrated into the for statement.

For example, the following For statement continually displays the value of the $i variable until you manually break out of the command by pressing CTRL+C.

```
$i = 1
for (;;){Write-Host $i}
```

You can add additional commands to the statement list so that the value of $i is incremented by 1 each time the loop is run, as the following example shows.

```
for (;;){$i++; Write-Host $i}
```

Until you break out of the command by pressing CTRL+C, this statement will continually display the value of the $i variable as it is incremented by 1 each time the loop is run.

Rather than change the value of the variable in the statement list part of the For statement, you can use the <repeat> portion of the For

statement instead, as follows.

```
$i=1
for (;;$i++){Write-Host $i}
```

This statement will still repeat indefinitely until you break out of the command by pressing CTRL+C.

By setting a condition (using the <condition> portion of the For statement), you can end the For loop when the condition evaluates to false. In the following example, the For loop runs while the value of $i is less than or equal to 10.

```
$i=1
for(;$i -le 10;$i++){Write-Host $i}
```

Instead of creating and initializing the variable outside the For statement, you can perform this task inside the For loop by using the <init> portion of the For statement.

```
for($i=1; $i -le 10; $i++){Write-Host $i}
```

You can use carriage returns instead of semicolons to delimit the <init>, <condition>, and <repeat> portions of the For statement. The following example shows the For statement syntax in this alternative form.

```
 for (<init>
<condition>
<repeat>){
<statement list>
}
```

This alternative form of the For statement works in Windows PowerShell script files and at the Windows PowerShell command prompt. However, it is easier to use the For statement syntax with semicolons when you enter interactive commands at the command prompt.

The For loop is more flexible than the Foreach loop because it allows you to increment values in an array or collection by using patterns. In the following example, the $i variable is incremented by 2 in the <repeat> portion of the for statement.

```
for ($i = 0; $i -ile 20; $i += 2) {Write-Host $i}
```

SEE ALSO
about_Comparison_Operators
about_Foreach

```
Name                    Category  Module            Synopsis
----                    --------  ------            --------
about_ForEach-Parallel  HelpFile                    Describes the ForEach -Parallel language construct
in
about_ForEach-Parallel  HelpFile                    Describes the ForEach -Parallel language construct
in
```

about_Foreach

SHORT DESCRIPTION
Describes a language command you can use to traverse all the items in a collection of items.

LONG DESCRIPTION
The Foreach statement (also known as a Foreach loop) is a language construct for stepping through (iterating) a series of values in a

collection of items.

The simplest and most typical type of collection to traverse is an array. Within a Foreach loop, it is common to run one or more commands against each item in an array.

Syntax

The following shows the ForEach syntax:

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

Simplified syntax

Starting in Windows PowerShell 3.0, syntax with language keywords such as Where and ForEach was simplified. Comparison operators that work on the members of a collection are treated as parameters. You can use a method on the members of a collection without containing it in a script block or adding the automatic variable "$_.". Consider the following two examples:

```
dir cert:\ -Recurse | foreach GetKeyAlgorithm
dir cert:\ -Recurse | foreach {$_.GetKeyAlgorithm()}
```

Although both commands work, the first returns results without using a script block or the $_. automatic variable. The method GetKeyAlgorithm is treated as a parameter of ForEach. The first command returns the same results, but without errors, because the simplified syntax does not attempt to return results for items for which the specified argument did not apply.

In this example, the Get-Process property Description is passed as a parameter argument of the ForEach statement. The results are the descriptions of active processes.

```
Get-Process | ForEach Description
```

The Foreach statement outside a command pipeline

The part of the Foreach statement enclosed in parenthesis represents a variable and a collection to iterate. Windows PowerShell creates the variable ($<item>) automatically when the Foreach loop runs. Prior to each iteration through the loop, the variable is set to a value in the collection. The block following a Foreach statement {<statement list>} contains a set of commands to execute against each item in a collection.

Examples

For example, the Foreach loop in the following example displays the values in the $letterArray array.

```
$letterArray = "a","b","c","d"
foreach ($letter in $letterArray)
{
   Write-Host $letter
}
```

In this example, the $letterArray array is created and initialized with the string values "a", "b", "c", and "d". The first time the Foreach statement runs, it sets the $letter variable equal to the first item in $letterArray ("a"). Then, it uses the Write-Host cmdlet to display the letter a. The next time through the loop, $letter is set to "b", and so on. After the Foreach loop displays the letter d, Windows PowerShell exits the loop.

The entire Foreach statement must appear on a single line to run it as a command at the Windows PowerShell command prompt. The entire Foreach statement does not have to appear on a single line if you place the command in a .ps1 script file instead.

Foreach statements can also be used together with cmdlets that return a collection of items. In the following example, the Foreach statement steps through the list of items that is returned by the Get-ChildItem cmdlet.

```
foreach ($file in Get-ChildItem)
{
   Write-Host $file
}
```

You can refine the example by using an If statement to limit the results that are returned. In the following example, the Foreach statement performs the same looping operation as the previous example, but it adds an If statement to limit the results to files that are greater than 100 kilobytes (KB):

```
foreach ($file in Get-ChildItem)
{
   if ($file.length -gt 100KB)
   {
      Write-Host $file
```

```
    }
}
```

In this example, the Foreach loop uses a property of the $file variable to perform a comparison operation ($file.length -gt 100KB). The $file variable contains all the properties in the object that is returned by the Get-ChildItem cmdlet. Therefore, you can return more than just a file name. In the next example, Windows PowerShell returns the length and the last access time inside the statement list:

```
foreach ($file in Get-ChildItem)
{
    if ($file.length -gt 100KB)
    {
        Write-Host $file
        Write-Host $file.length
        Write-Host $file.lastaccesstime
    }
}
```

In this example, you are not limited to running a single command in a statement list.

You can also use a variable outside a Foreach loop and increment the variable inside the loop. The following example counts files over 100 KB in size:

```
$i = 0
foreach ($file in Get-ChildItem)
{
    if ($file.length -gt 100KB)
    {
        Write-Host $file "file size:" ($file.length /
1024).ToString("F0") KB
        $i = $i + 1
    }
}

if ($i -ne 0)
{
    Write-Host
    Write-Host $i " file(s) over 100 KB in the current
directory."}
```

```
else
{
   Write-Host "No files greater than 100 KB in the current
directory."
}
```

In the preceding example, the $i variable is set to 0 outside the loop, and the variable is incremented inside the loop for each file that is found that is larger than 100 KB. When the loop exits, an If statement evaluates the value of $i to display a count of all the files over 100 KB. Or, it displays a message stating that no files over 100 KB were found.

The previous example also demonstrates how to format the file length results:

```
($file.length / 1024).ToString("F0")
```

The value is divided by 1,024 to show the results in kilobytes rather than bytes, and the resulting value is then formatted using the fixed-point format specifier to remove any decimal values from the result. The 0 makes the format specifier show no decimal places.

The Foreach Statement Inside a Command Pipeline

When Foreach appears in a command pipeline, Windows PowerShell uses the foreach alias, which calls the ForEach-Object command. When you use the foreach alias in a command pipeline, you do not include the ($<item> in $<collection>) syntax as you do with the Foreach statement. This is because the prior command in the pipeline provides this information. The syntax of the foreach alias when used in a command pipeline is as follows:

```
<command> | foreach {<command_block>}
```

For example, the Foreach loop in the following command displays processes whose working set (memory usage) is greater than 20 megabytes (MB).

The Get-Process command gets all of the processes on the computer. The Foreach alias performs the commands in the script block on each process in sequence.

The IF statement selects processes with a working set (WS) greater than 20 megabytes. The Write-Host cmdlet writes the name of the process followed by a colon. It divides the working set value, which is stored in bytes by 1 megabyte to get the working set value in megabytes. Then it converts the result from a double to a string. It displays the value as a fixed point number with zero decimals (F0), a space separator (" "), and then "MB".

```
Write-Host "Processes with working sets greater than 20 MB."
Get-Process | foreach {
  if ($_.WS -gt 20MB)
  { Write-Host $_.name ": " ($_.WS/1MB).ToString("F0") MB -Separator ""}
}
```

The foreach alias also supports beginning command blocks, middle command blocks, and end command blocks. The beginning and end command blocks run once, and the middle command block runs every time the Foreach loop steps through a collection or array.

The syntax of the foreach alias when used in a command pipeline with a beginning, middle, and ending set of command blocks is as follows:

```
<command> | foreach {<beginning command_block>}{<middle
command_block>}{<ending command_block>}
```

The following example demonstrates the use of the beginning, middle, and end command blocks.

```
Get-ChildItem | foreach {
$fileCount = $directoryCount = 0}{
if ($_.PsIsContainer) {$directoryCount++} else {$fileCount++}}{
"$directoryCount directories and $fileCount files"}
```

The beginning block creates and initializes two variables to 0:

```
{$fileCount = $directoryCount = 0}
```

The middle block evaluates whether each item returned by Get-ChildItem is a directory or a file:

```
{if ($_.PsIsContainer) {$directoryCount++} else {$fileCount++}}
```

If the item that is returned is a directory, the $directoryCount variable is incremented by 1. If the item is not a directory, the $fileCount variable is incremented by 1. The ending block runs after the middle block completes its looping operation and then returns the results of the operation:

```
{"$directoryCount directories and $fileCount files"}
```

By using the beginning, middle, and ending command block structure and the pipeline operator, you can rewrite the earlier example to find any files that are greater than 100 KB, as follows:

```
Get-ChildItem | foreach{
   $i = 0}{
   if ($_.length -gt 100KB)
   {
      Write-Host $_.name "file size:" ($_.length /
1024).ToString("F0") KB
      $i++
   }
   }{
   if ($i -ne 0)
   {
      Write-Host
      Write-Host "$i file(s) over 100 KB in the current
directory."
   }
   else
   {
   Write-Host "No files greater than 100 KB in the current
directory."}
   }
```

The following example, a function which returns the functions that are used in scripts and script modules, demonstrates how to use the MoveNext method (which works similarly to "skip X" on a For loop) and the Current property of the $foreach variable inside of a foreach script block, even if there are unusually- or inconsistently-spaced function definitions that span multiple lines to declare the function name. The example also works if there are comments in the functions used in a script or script module.

```
function Get-FunctionPosition {
  [CmdletBinding()]
  [OutputType('FunctionPosition')]
  param(
    [Parameter(Position=0, Mandatory, ValueFromPipeline, ValueFromPipelineByPropertyName)]
    [ValidateNotNullOrEmpty()]
    [Alias('PSPath')]
    [System.String[]]
    $Path
  )
  process {
    try {
      $filesToProcess = if ($_ -is [System.IO.FileSystemInfo]) {
        $_
      } else {
        Get-Item -Path $Path
      }
      foreach ($item in $filesToProcess) {
        if ($item.PSIsContainer -or $item.Extension -notin @('.ps1','.psm1')) {
          continue
        }
        $tokens = $errors = $null
        $ast =
[System.Management.Automation.Language.Parser]::ParseFile($item.FullName,([REF]$tokens),([REF]$er
rors))
        if ($errors) {
          Write-Warning "File '$($item.FullName)' has $($errors.Count) parser errors."
        }
        :tokenLoop foreach ($token in $tokens) {
          if ($token.Kind -ne 'Function') {
            continue
          }
          $position = $token.Extent.StartLineNumber
          do {
            if (-not $foreach.MoveNext()) {
              break tokenLoop
            }
            $token = $foreach.Current
          } until ($token.Kind -in @('Generic','Identifier'))
          $functionPosition = [pscustomobject]@{
                Name = $token.Text
            LineNumber = $position
                Path = $item.FullName
          }
          Add-Member -InputObject $functionPosition -TypeName FunctionPosition -PassThru
        }
      }
    } catch {
```

```
            throw
        }
    }
}
```

SEE ALSO

about_Automatic_Variables
about_If
Foreach-Object

about_Format.ps1xml

SHORT DESCRIPTION

The Format.ps1xml files in Windows PowerShell define the default display of objects in the Windows PowerShell console. You can create your own Format.ps1xml files to change the display of objects or to define default displays for new object types that you create in Windows PowerShell.

LONG DESCRIPTION

The Format.ps1xml files in Windows PowerShell define the default display of objects in Windows PowerShell. You can create your own Format.ps1xml files to change the display of objects or to define default displays for new object types that you create in Windows PowerShell.

When Windows PowerShell displays an object, it uses the data in structured formatting files to determine the default display of the object. The data in the formatting files determines whether the object is rendered in a table or in a list, and it determines which properties are displayed by default.

The formatting affects the display only. It does not affect which object

properties are passed down the pipeline or how they are passed. Format.ps1xml files cannot be used to customize the output format for hashtables.

Windows PowerShell includes seven formatting files. These files are located in the installation directory ($pshome). Each file defines the display of a group of Microsoft .NET Framework objects:

Certificate.Format.ps1xml
: Objects in the Certificate store, such as X.509 certificates and certificate stores.

DotNetTypes.Format.ps1xml
: Other .NET Framework types, such as CultureInfo, FileVersionInfo, and EventLogEntry objects.

FileSystem.Format.ps1xml
: File system objects, such as files and directories.

Help.Format.ps1xml
: Help views, such as detailed and full views, parameters, and examples.

PowerShellCore.format.ps1xml
: Objects generated by Windows PowerShell core cmdlets, such as Get-Member and Get-History.

PowerShellTrace.format.ps1xml
: Trace objects, such as those generated by the Trace-Command cmdlet.

Registry.format.ps1xml
: Registry objects, such as keys and entries.

A formatting file can define four different views of each object: table, list, wide, and custom. For example, when the output of a Get-ChildItem command is piped to a Format-List command, Format-List uses the view in the FileSystem.format.ps1xml file to determine how to display the file and folder objects as a list.

When a formatting file includes more than one view of an object, Windows PowerShell applies the first view that it finds.

In a Format.ps1xml file, a view is defined by a set of XML tags that describe the name of the view, the type of object to which it can be applied, the column headers, and the properties that are displayed in the body of the view. The format in Format.ps1xml files is applied

just before the data is presented to the user.

Creating New Format.ps1xml Files

The .ps1xml files that are installed with Windows PowerShell are digitally signed to prevent tampering because the formatting can include script blocks. Therefore, to change the display format of an existing object view, or to add views for new objects, create your own Format.ps1xml files, and then add them to your Windows PowerShell session.

To create a new file, copy an existing Format.ps1xml file. The new file can have any name, but it must have a .ps1xml file name extension. 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 change the formatting of a current view, locate the view in the formatting file, and then use the tags to change the view. To create a view for a new object type, create a new view, or use an existing view as a model. (The tags are described in the next section of this topic.) You can then delete all the other views in the file so that the changes are obvious to anyone examining the file.

When you have saved the changes, use the Update-FormatData cmdlet to add the new file to your Windows PowerShell session. If you want your view to take precedence over a view defined in the built-in files, use the PrependData parameter of Update-FormatData. Update-FormatData affects only the current session. To make the change to all future sessions, add the Update-FormatData command to your Windows PowerShell profile.

Example: Adding Calendar Data to Culture Objects

This example shows how to change the formatting of the culture objects (System.Globalization.CultureInfo) generated by the Get-Culture cmdlet. The commands in the example add the calendar property to the default table view display of culture objects.

The first step is to find the Format.ps1xml file that contains the current view of the culture objects. The following Select-String command finds the file:

```
select-string -path $pshome\*format.ps1xml `
        -pattern System.Globalization.CultureInfo
```

This command reveals that the definition is in the DotNetTypes.Format.ps1xml file.

The next command copies the file contents to a new file, MyDotNetTypes.Format.ps1xml.

```
copy-item DotNetTypes.Format.ps1xml MyDotNetTypes.Format.ps1xml
```

Next, open the MyDotNetTypes.Format.ps1xml file in any XML or text editor, such as Notepad. Find the System.Globalization.CultureInfo object section. The following XML defines the views of the CultureInfo object. The object has only a TableControl view.

```
<View>
  <Name>System.Globalization.CultureInfo</Name>
  <ViewSelectedBy>
    <TypeName>Deserialized.System.Globalization.CultureInfo</TypeName>
    <TypeName>System.Globalization.CultureInfo</TypeName>
  </ViewSelectedBy>

  <TableControl>
    <TableHeaders>
      <TableColumnHeader>
        <Width>16</Width>
      </TableColumnHeader>
      <TableColumnHeader>
        <Width>16</Width>
      </TableColumnHeader>
      <TableColumnHeader/>
    </TableHeaders>
    <TableRowEntries>
      <TableRowEntry>
        <TableColumnItems>
          <TableColumnItem>
            <PropertyName>LCID</PropertyName>
          </TableColumnItem>
          <TableColumnItem>
            <PropertyName>Name</PropertyName>
          </TableColumnItem>
```

```
            <TableColumnItem>
              <PropertyName>DisplayName</PropertyName>
            </TableColumnItem>
          </TableColumnItems>
        </TableRowEntry>
      </TableRowEntries>
  </TableControl>
</View>
```

Delete the remainder of the file, except for the opening <?XML>, <Configuration>, and <ViewDefinitions> tags and the closing <ViewDefintions> and <Configuration> tags. You must also delete the digital signature whenever you change the file.

```
<?xml version="1.0" encoding="utf-8" ?>
<Configuration>
  <ViewDefinitions>
    <View>
      <Name>System.Globalization.CultureInfo</Name>
      <ViewSelectedBy>
        <TypeName>Deserialized.System.Globalization.CultureInfo</TypeName>
        <TypeName>System.Globalization.CultureInfo</TypeName>
      </ViewSelectedBy>

      <TableControl>
        <TableHeaders>
          <TableColumnHeader>
            <Width>16</Width>
          </TableColumnHeader>
          <TableColumnHeader>
            <Width>16</Width>
          </TableColumnHeader>
          <TableColumnHeader/>
        </TableHeaders>
        <TableRowEntries>
          <TableRowEntry>
            <TableColumnItems>
              <TableColumnItem>
                <PropertyName>LCID</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>Name</PropertyName>
              </TableColumnItem>
              <TableColumnItem>
                <PropertyName>DisplayName</PropertyName>
              </TableColumnItem>
```

```
            </TableColumnItems>
          </TableRowEntry>
        </TableRowEntries>
      </TableControl>
    </View>
  </ViewDefinitions>
</Configuration>
```

Next, create a new column for the Calendar property by adding a new set of <TableColumnHeader> tags. The value of the Calendar property can be long, so a value of 45 characters is used, as follows:

```
      <TableControl>
        <TableHeaders>
          <TableColumnHeader>
            <Width>16</Width>
          </TableColumnHeader>
          <TableColumnHeader>
            <Width>16</Width>
          </TableColumnHeader>

          <TableColumnHeader>
            <Width>45</Width>
          </TableColumnHeader>

          <TableColumnHeader/>
        </TableHeaders>
```

Now, add a new column item in the table rows, as follows:

```
      <TableRowEntries>
        <TableRowEntry>
          <TableColumnItems>
            <TableColumnItem>
              <PropertyName>LCID</PropertyName>
            </TableColumnItem>
            <TableColumnItem>
              <PropertyName>Name</PropertyName>
            </TableColumnItem>

            <TableColumnItem>
              <PropertyName>Calendar</PropertyName>
            </TableColumnItem>
```

```
            <TableColumnItem>
              <PropertyName>DisplayName</PropertyName>
            </TableColumnItem>
          </TableColumnItems>
        </TableRowEntry>
      </TableRowEntries>
```

After saving the file and closing it, use an Update-FormatData command, such as the following command, to add the new format file to the current session. The command uses the PrependData parameter to place the new file in a higher precedence order than the original file. (For more information about Update-FormatData, type "get-help update-formatdata".)

```
update-formatdata -prependpath $pshome\MyDotNetTypes.format.ps1xml
```

To test the change, type "get-culture", and then review the output, which includes the Calendar property.

```
C:\PS> get-culture

LCID Name  Calendar                         DisplayName
---- ----  --------                         -----------
1033 en-US System.Globalization.GregorianCalendar English (United States)
```

The XML in Format.ps1xml Files

The ViewDefinitions section of each Format.ps1xml file contains the <View> tags that define each view. A typical <View> tag includes the following tags:

<Name>
The <Name> tag identifies the name of the view.

<ViewSelectedBy>
The <ViewSelectedBy> tag specifies the object type or types to which the view applies.

<GroupBy>
The <GroupBy> tag specifies how items in the view will be

combined in groups.

<TableControl>
<ListControl>
<WideControl>
<CustomControl>
These tags contain the tags that specify how each item will be displayed.

The <ViewSelectedBy> tag can contain a <TypeName> tag for each object type to which the view applies. Or, it can contain a <SelectionSetName> tag that references a selection set that is defined elsewhere by using a <SelectionSet> tag.

The <GroupBy> tag contains a <PropertyName> tag that specifies the object property by which items are to be grouped. It also contains either a <Label> tag that specifies a string to be used as a label for each group or a <CustomControlName> tag that references a custom control defined elsewhere using a <Control> tag. The <Control> tag contains a <Name> tag and a <CustomControl> tag.

The <TableControl> tag typically contains <TableHeaders> and <TableRowEntries> tags that define the formatting for the table's heads and rows. The <TableHeaders> tag typically contains <TableColumnHeader> tags that contain <Label>, <Width>, and <Alignment> tags. The <TableRowEntries> tag contains <TableRowEntry> tags for each row in the table. The <TableRowEntry> tag contains a <TableColumnItems> tag that contains a <TableColumnItem> tag for each column in the row. Typically, the <TableColumnItem> tag contains either a <PropertyName> tag that identifies the object property to be displayed in the defined location, or a <ScriptBlock> tag that contains script code that calculates a result that is to be displayed in the location.

Note: Script blocks can also be used elsewhere in locations where calculated results can be useful.

The <TableColumnItem> tag can also contain a <FormatString> tag that specifies how the property or the calculated results will be displayed.

The <ListControl> tag typically contains a <ListEntries> tag. The <ListEntries> tag contains a <ListItems> tag. The <ListItems> tag

contains <ListItem> tags, which contain <PropertyName> tags.
The <PropertyName> tags specify the object property to be displayed at
the specified location in the list. If the view selection is defined
using a selection set, the <ListControl> tag can also contain an
<EntrySelectedBy> tag that contains one or more <TypeName> tags. These
<TypeName> tags specify the object type that the <ListControl> tag is
intended to display.

The <WideControl> tag typically contains a <WideEntries> tag. The
<WideEntries> tag contains one or more <WideEntry> tags. A <WideEntry>
tag typically contains a <PropertyName> tag that specifies the property
to be displayed at the specified location in the view. The <PropertyName>
tag can contain a <FormatString> tag that specifies how the property is
to be displayed.

The <CustomControl> tag lets you use a script block to define a format.
A <CustomControl> tag typically contains a <CustomEntries> tag that contains
multiple <CustomEntry> tags. Each <CustomEntry> tag contains a <CustomItem>
tags that can contain a variety of tags that specify contents and formatting
of the specified location in the view, including <Text>, <Indentation>,
<ExpressionBinding>, and <NewLine> tags.

Update-FormatData

To load your Format.ps1xml files into a Windows PowerShell session, use
the Update-FormatData cmdlet. If you want the views in your file to
take precedence over the views in the built-in Format.ps1xml file, use
the PrependData parameter of Update-FormatData. Update-FormatData affects
only the current session. To make the change to all future sessions, add
the Update-FormatData command to your Windows PowerShell profile.

Default Displays in Types.ps1xml

The default displays of some basic object types are defined in the
Types.ps1xml file in the $pshome directory. The nodes are named
PsStandardMembers, and the subnodes use one of the following tags:

<DefaultDisplayProperty>
<DefaultDisplayPropertySet>
<DefaultKeyPropertySet>

For more information, type the following command:

get-help about_types.ps1xml

Tracing Format.ps1xml File Use

To detect errors in the loading or application of Format.ps1xml files, use the Trace-Command cmdlet with any of the following format components as the value of the Name parameter:

FormatFileLoading
UpdateFormatData
FormatViewBinding

For more information, type the following commands:

get-help trace-command
get-help get-tracesource

Signing a Format.ps1xml File

To protect the users of your Format.ps1xml file, sign the file using a digital signature. For more information, type:

get-help about_signing

SEE ALSO
Update-FormatData
Trace-Command
Get-TraceSource

TOPIC

about_Functions

SHORT DESCRIPTION

Describes how to create and use functions in Windows PowerShell.

LONG DESCRIPTION

A function is a list of Windows PowerShell statements that has a name that you assign. When you run a function, you type the function name. The statements in the list run as if you had typed them at the command prompt.

Functions can be as simple as:

```
function Get-PowerShellProcess {Get-Process PowerShell}
```

or as complex as a cmdlet or an application program.

Like cmdlets, functions can have parameters. The parameters can be named, positional, switch, or dynamic parameters. Function parameters can be read from the command line or from the pipeline.

Functions can return values that can be displayed, assigned to variables, or passed to other functions or cmdlets.

The function's statement list can contain different types of statement lists with the keywords Begin, Process, and End. These statement lists handle input from the pipeline differently.

A filter is a special kind of function that uses the Filter keyword.

Functions can also act like cmdlets. You can create a function that works just like a cmdlet without using C# programming. For more information, see about_Functions_Advanced (http://go.microsoft.com/fwlink/?LinkID=144511).

Syntax

The following is the syntax for a function:

```
function [<scope:>]<name> [([type]$parameter1[,[type]$parameter2])]
{
   param([type]$parameter1 [,[type]$parameter2])

   dynamicparam {<statement list>}

   begin {<statement list>}
   process {<statement list>}
```

```
    end {<statement list>}
}
```

A function includes the following items:

- A Function keyword
- A scope (optional)
- A name that you select
- Any number of named parameters (optional)
- One or more Windows PowerShell commands enclosed in braces ({})

For more information about the Dynamicparam keyword and dynamic parameters in functions, see about_Functions_Advanced_Parameters.

Simple Functions

Functions do not have to be complicated to be useful. The simplest functions have the following format:

```
function <function-name> {statements}
```

For example, the following function starts Windows PowerShell with the Run as Administrator option.

```
function Start-PSAdmin {Start-Process PowerShell -Verb RunAs}
```

To use the function, type: Start-PSAdmin

To add statements to the function, use a semi-colon (;) to separate the statements, or type each statement on a separate line.

For example, the following function finds all .jpg files in the current user's directories that were changed after the start date.

```
function Get-NewPix
{
   $start = Get-Date -Month 1 -Day 1 -Year 2010
   $allpix = Get-ChildItem -Path $env:UserProfile\*.jpg -Recurse
   $allpix | where {$_.LastWriteTime -gt $Start}
}
```

You can create a toolbox of useful small functions. Add these functions to your Windows PowerShell profile, as described in about_Profiles and later in this topic.

Function Names

You can assign any name to a function, but functions that you share with others should follow the naming rules that have been established for all Windows PowerShell commands.

Functions names should consist of a verb-noun pair in which the verb identifies the action that the function performs and the noun identifies the item on which the cmdlet performs its action.

Functions should use the standard verbs that have been approved for all Windows PowerShell commands. These verbs help us to keep our command names simple, consistent, and easy for users to understand.

For more information about the standard Windows PowerShell verbs, see "Cmdlet Verbs" on MSDN at http://go.microsoft.com/fwlink/?LinkID=160773.

Functions with Parameters

You can use parameters with functions, including named parameters, positional parameters, switch parameters, and dynamic parameters. For more information about dynamic parameters in functions, see about_Functions_Advanced_Parameters (http://go.microsoft.com/fwlink/?LinkID=135173).

Named Parameters

You can define any number of named parameters. You can include a default value for named parameters, as described later in this topic.

You can define parameters inside the braces using the Param keyword, as shown in the following sample syntax:

```
function <name> {
    param ([type]$parameter1[,[type]$parameter2])
    <statement list>
}
```

You can also define parameters outside the braces without the Param keyword, as shown in the following sample syntax:

```
function <name> [([type]$parameter1[,[type]$parameter2])] {
   <statement list>
}
```

There is no difference between these two methods. Use the method that

you prefer.

When you run the function, the value you supply for a parameter is assigned to a variable that contains the parameter name. The value of that variable can be used in the function.

The following example is a function called Get-SmallFiles. This function has a $size parameter. The function displays all the files that are smaller than the value of the $size parameter, and it excludes directories:

```
function Get-SmallFiles {
   param ($size)
   Get-ChildItem c:\ | where {$_.Length -lt $Size -and !$_.PSIsContainer}
}
```

In the function, you can use the $size variable, which is the name defined for the parameter.

To use this function, type the following command:

```
C:\PS> function Get-SmallFiles –Size 50
```

You can also enter a value for a named parameter without the parameter name. For example, the following command gives the same result as a command that names the Size parameter:

```
C:\PS> function Get-SmallFiles 50
```

To define a default value for a parameter, type an equal sign and the value after the parameter name, as shown in the following variation of the Get-SmallFiles example:

```
function Get-SmallFiles ($size = 100) {
   Get-ChildItem c:\ | where {$_.Length -lt $Size -and !$_.PSIsContainer}
}
```

If you type "Get-SmallFiles" without a value, the function assigns 100 to $size. If you provide a value, the function uses that value.

Optionally, you can provide a brief help string that describes the default value of your parameter, by adding the PSDefaultValue attribute to the description of your parameter, and specifying the Help property of PSDefaultValue. To provide a help string that describes the default value (100) of the Size parameter in the

Get-SmallFiles function, add the PSDefaultValue attribute as shown in the following example.

```
function Get-SmallFiles {
   param (
   [PSDefaultValue(Help = '100')]
   $size = 100
   )
```

For more information about the PSDefaultValue attribute class, see PSDefaultValue Attribute Members on MSDN.

(http://msdn.microsoft.com/library/windows/desktop/system.management.automation.psdefaultvalueattribute_members(v=vs.85).aspx

Positional Parameters

A positional parameter is a parameter without a parameter name. Windows PowerShell uses the parameter value order to associate each parameter value with a parameter in the function.

When you use positional parameters, type one or more values after the function name. Positional parameter values are assigned to the $args array variable. The value that follows the function name is assigned to the first position in the $args array, $args[0].

The following Get-Extension function adds the .txt file name extension to a file name that you supply:

```
function Get-Extension {
   $name = $args[0] + ".txt"
   $name
}

C:\PS> Get-Extension myTextFile
myTextFile.txt
```

Switch Parameters

A switch is a parameter that does not require a value. Instead, you type the function name followed by the name of the switch parameter.

To define a switch parameter, specify the type [switch] before the parameter name, as shown in the following example:

```
function Switch-Item {
   param ([switch]$on)
   if ($on) { "Switch on" }
   else { "Switch off" }
}
```

When you type the On switch parameter after the function name, the function displays "Switch on". Without the switch parameter, it displays "Switch off".

```
C:\PS> Switch-Item -on
Switch on

C:\PS> Switch-Item
Switch off
```

You can also assign a Boolean value to a switch when you run the function, as shown in the following example:

```
C:\PS> Switch-Item -on:$true
Switch on

C:\PS> Switch-Item -on:$false
Switch off
```

Using Splatting to Represent Command Parameters

You can use splatting to represent the parameters of a command. This feature is introduced in Windows PowerShell 3.0.

Use this technique in functions that call commands in the session. You do not need to declare or enumerate the command parameters, or change the function when command parameters change.

The following sample function calls the Get-Command cmdlet. The command uses @Args to represent the parameters of Get-Command.

```
function Get-MyCommand { Get-Command @Args }
```

You can use all of the parameters of Get-Command when you call the Get-MyCommand function. The parameters and parameter values are passed to the command using @Args.

```
PS C:\>Get-MyCommand -Name Get-ChildItem
CommandType     Name              ModuleName
-----------     ----              ----------
Cmdlet          Get-ChildItem     Microsoft.PowerShell.Management
```

The @Args feature uses the $Args automatic parameter, which represents undeclared cmdlet parameters and values from remaining arguments.

For more information about splatting, see about_Splatting (http://go.microsoft.com/fwlink/?LinkId=262720).

Piping Objects to Functions

Any function can take input from the pipeline. You can control how a function processes input from the pipeline using Begin, Process, and End keywords. The following sample syntax shows the three keywords:

```
function <name> {
   begin {<statement list>}
   process {<statement list>}
   end {<statement list>}
}
```

The Begin statement list runs one time only, at the beginning of the function.

The Process statement list runs one time for each object in the pipeline. While the Process block is running, each pipeline object is assigned to the $_ automatic variable, one pipeline object at a time.

After the function receives all the objects in the pipeline, the End statement list runs one time. If no Begin, Process, or End keywords are used, all the statements are treated like an End statement list.

The following function uses the Process keyword. The function displays examples from the pipeline:

```
function Get-Pipeline
{
   process {"The value is: $_"}
}
```

To demonstrate this function, enter an list of numbers separated by commas, as shown in the following example:

```
C:\PS> 1,2,4 | Get-Pipeline
The value is: 1
The value is: 2
The value is: 4
```

When you use a function in a pipeline, the objects piped to the function are assigned to the $input automatic variable. The function runs statements with the Begin keyword before any objects come from the pipeline. The function runs statements with the End keyword after all the objects have been received from the pipeline.

The following example shows the $input automatic variable with Begin and End keywords.

```
function Get-PipelineBeginEnd
{
   begin {"Begin: The input is $input"}
   end {"End:   The input is $input" }
}
```

If this function is run by using the pipeline, it displays the following results:

```
C:\PS> 1,2,4 | Get-PipelineBeginEnd
Begin: The input is
End:   The input is 1 2 4
```

When the Begin statement runs, the function does not have the input from the pipeline. The End statement runs after the function has the values.

If the function has a Process keyword, the function reads the data in $input. The following example has a Process statement list:

```
function Get-PipelineInput
{
   process {"Processing:  $_ " }
   end {"End:   The input is: $input" }
}
```

In this example, each object that is piped to the function is sent to the Process statement list. The Process statements run on each object, one object at a time. The $input automatic variable is empty when the function reaches the End keyword.

```
C:\PS> 1,2,4 | Get-PipelineInput
Processing:  1
Processing:  2
Processing:  4
End:   The input is:
```

Filters

A filter is a type of function that runs on each object in the pipeline. A filter resembles a function with all its statements in a Process block.

The syntax of a filter is as follows:

```
filter [<scope:>]<name> {<statement list>}
```

The following filter takes log entries from the pipeline and then displays either the whole entry or only the message portion of the entry:

```
filter Get-ErrorLog ([switch]$message)
{
   if ($message) { out-host -inputobject $_.Message }
   else { $_ }
}
```

Function Scope

A function exists in the scope in which it was created.

If a function is part of a script, the function is available to statements within that script. By default, a function in a script is not available at the command prompt.

You can specify the scope of a function. For example, the function is added to the global scope in the following example:

```
function global:Get-DependentSvs { Get-Service |
  where {$_.DependentServices} }
```

When a function is in the global scope, you can use the function in scripts, in functions, and at the command line.

Functions normally create a scope. The items created in a function, such as variables, exist only in the function scope.

For more information about scope in Windows PowerShell, see about_Scopes (http://go.microsoft.com/fwlink/?LinkID=113260).

Finding and Managing Functions Using the Function: Drive

All the functions and filters in Windows PowerShell are automatically stored in the Function: drive. This drive is exposed by the Windows PowerShell Function provider.

When referring to the Function: drive, type a colon after Function, just as you would do when referencing the C or D drive of a computer.

The following command displays all the functions in the current session of Windows PowerShell:

Get-ChildItem function:

The commands in the function are stored as a script block in the definition property of the function. For example, to display the commands in the Help function that comes with Windows PowerShell, type:

(Get-ChildItem function:help).Definition

For more information about the Function: drive, see the help topic for the Function provider. Type "Get-Help Function" or view it in the TechNet Library at http://go.microsoft.com/fwlink/?LinkID=113436.

Reusing Functions in New Sessions

When you type a function at the Windows PowerShell command prompt, the function becomes part of the current session. It is available until the session ends.

To use your function in all Windows PowerShell sessions, add the function to your Windows PowerShell profile. For more information about profiles, see about_Profiles (http://go.microsoft.com/fwlink/?LinkID=113729).

You can also save your function in a Windows PowerShell script file. Type your function in a text file, and then save the file with the .ps1 file name extension.

Writing Help for Functions

The Get-Help cmdlet gets help for functions, as well as for cmdlets, providers, and scripts. To get help for a function, type Get-Help followed by the function name.

For example, to get help for the Get-MyDisks function, type:

Get-Help Get-MyDisks

You can write help for a function by using either of the two following methods:

-- Comment-Based Help for Functions

Create a help topic by using special keywords in the comments. To create comment-based help for a function, the comments must be placed at the beginning or end of the function body or on the lines preceding the function keyword. For more information about comment-based help, see about_Comment_Based_Help.

-- XML-Based Help for Functions

Create an XML-based help topic, such as the type that is typically created for cmdlets. XML-based help is required if you are localizing help topics into multiple languages.

To associate the function with the XML-based help topic, use the .ExternalHelp comment-based help keyword. Without this keyword, Get-Help cannot find the function help topic and calls to Get-Help for the function return only auto-generated help.

For more information about the ExternalHelp keyword, see about_Comment_Based_Help. For more information about XML-based help, see "How to Write Cmdlet Help" in MSDN.

SEE ALSO

about_Automatic_Variables
about_Comment_Based_Help
about_Functions_Advanced
about_Functions_Advanced_Methods
about_Functions_Advanced_Parameters
about_Functions_CmdletBindingAttribute
about_Functions_OutputTypeAttribute
about_Parameters
about_Profiles
about_Scopes
about_Script_Blocks
Function (provider)

TOPIC

about_Functions_Advanced

SHORT DESCRIPTION

Introduces advanced functions that act similar to cmdlets.

LONG DESCRIPTION

Advanced functions allow you to write functions that can perform operations that are similar to the operations you can perform with cmdlets. Advanced functions are helpful when you want to quickly write a function without having to write a compiled cmdlet using a Microsoft .NET Framework language. These functions are also helpful when you want to restrict the functionality of a compiled cmdlet or when you want to write a function that is similar to a compiled cmdlet.

There is a difference between authoring a compiled cmdlet and an advanced function. Compiled cmdlets are .NET Framework classes that must be written in a .NET Framework language such as C#. In contrast, advanced functions are written in the Windows PowerShell script language in the same way that other functions or script blocks are written.

Advanced functions use the CmdletBinding attribute to identify them as functions that act similar to cmdlets. The CmdletBinding attribute is similar to the Cmdlet attribute that is used in compiled cmdlet classes to identify the class as a cmdlet. For more information about this attribute, see about_Functions_CmdletBindingAttribute.

The following example shows a function that accepts a name and then prints a greeting using the supplied name. Also notice that this function defines a name that includes a verb (Send) and noun (Greeting) pair similar to the verb-noun pair of a compiled cmdlet. However, functions are not required to have a verb-noun name.

```
function Send-Greeting
{
 [CmdletBinding()]
 Param(
    [Parameter(Mandatory=$true)]
    [string] $Name
 )
 Process
 {
  write-host ("Hello " + $Name + "!")
 }
}
```

The parameters of the function are declared by using the Parameter attribute. This attribute can be used alone, or it can be combined with the Alias attribute or with several other parameter validation attributes. For more information about how to declare parameters (including dynamic parameters that are added at runtime), see about_Functions_Advanced_Parameters.

The actual work of the previous function is performed in the Process block, which is equivalent to the ProcessingRecord method that is used by compiled cmdlets to process the data that is passed to the cmdlet. This block, along with the Begin and End blocks, is described in the about_Functions_Advanced_Methods topic.

Advanced functions differ from compiled cmdlets in the following ways:

- Advanced function parameter binding does not throw an exception when an array of strings is bound to a Boolean parameter.

- The ValidateSet attribute and the ValidatePattern attribute cannot pass named parameters.

- Advanced functions cannot be used in transactions.

SEE ALSO

about_Functions
about_Functions_Advanced_Methods
about_Functions_Advanced_Parameters
about_Functions_CmdletBindingAttribute
about_Functions_OutputTypeAttribute
Windows PowerShell Cmdlets (http://go.microsoft.com/fwlink/?LinkID=135279)

TOPIC

about_Functions_Advanced_Methods

SHORT DESCRIPTION

Describes how functions that specify the CmdletBinding attribute can use the methods and properties that are available to compiled cmdlets.

LONG DESCRIPTION

Functions that specify the CmdletBinding attribute can access a number of methods and properties through the $pscmdlet variable. These methods include the following methods:

- Input-processing methods that compiled cmdlets use to do their work.
- The ShouldProcess and ShouldContinue methods that are used to get user feedback before an action is performed.
- The ThrowTerminatingError method for generating error records.
- Several Write methods that return different types of output.

All the methods and properties of the PSCmdlet class are available to advanced functions. For more information about these methods and properties, see System.Management.Automation.PSCmdlet in the MSDN (Microsoft Developer Network) library at http://go.microsoft.com/fwlink/?LinkId=142139.

Input Processing Methods

The methods described in this section are referred to as the input processing methods. For functions, these three methods are represented by the Begin, Process, and End blocks of the function. Each function must include one or more of these blocks. The Windows PowerShell runtime uses the code within these blocks when it is running a function. (These blocks are also available to functions that do not use the CmdletBinding attribute.)

Begin

This block is used to provide optional one-time preprocessing for the function. The Windows PowerShell runtime uses the code in this block one time for each instance of the function in the pipeline.

Process

This block is used to provide record-by-record processing for the function. This block might be used any number of times, or not at all, depending on the input to the function. For example, if the function is the first command in the pipeline, the Process block will be used one time. If the function is not the first command in the pipeline, the Process block is used one time for every input that the function receives from the pipeline. If there is no pipeline input, the Process block is not used.

This block must be defined if a function parameter is set to accept pipeline input. If this block is not defined and the parameter accepts input from the pipeline, the function will miss the values that are passed to the function through the pipeline.

Also, when the function supports confirmation requests (when the SupportsShouldProcess parameter of the Parameter attribute is set to $True), the call to the ShouldProcess method must be made from within the Process block.

End

This block is used to provide optional one-time post-processing for the function.

The following example shows the outline of a function that contains a Begin block for one-time preprocessing, a Process block for multiple record processing, and an End block for one-time post-processing.

```
Function Test-ScriptCmdlet
{
 [CmdletBinding(SupportsShouldProcess=$True)]
 Param ($Parameter1)
 Begin{}
 Process{}
 End{}
}
```

Confirmation Methods

ShouldProcess

This method is called to request confirmation from the user before the function performs an action that would change the system. The function can continue based on the Boolean value returned by the method. This method can be called only from within the Process{} block of the function. And, the CmdletBinding attribute must declare that the function supports ShouldProcess (as shown in the previous example).

For more information about this method, see

System.Management.Automation.Cmdlet.ShouldProcess in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142142.

For more information about how to request confirmation, see "Requesting Confirmation" in the MSDN library at http://go.microsoft.com/fwlink/?LinkID=136658.

ShouldContinue

This method is called to request a second confirmation message. It should be called when the ShouldProcess method returns $true. For more information about this method, see System.Management.Automation.Cmdlet.ShouldContinue in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142143.

Error Methods

Functions can call two different methods when errors occur. When a nonterminating error occurs, the function should call the WriteError method, which is described in the "Write Methods" section. When a terminating error occurs and the function cannot continue, it should call the ThrowTerminatingError method. You can also use the Throw statement for terminating errors and the Write-Error cmdlet for nonterminating errors.

For more information, see System.Management.Automation.Cmdlet.ThrowTerminatingError in the MSDN libray at http://go.microsoft.com/fwlink/?LinkId=142144.

Write Methods

A function can call the following methods to return different types of output. Notice that not all the output goes to the next command in the pipeline. You can also use the various Write cmdlets, such as Write-Error.

WriteCommandDetail

For information about the WriteCommandDetails method, see System.Management.Automation.Cmdlet.WriteCommandDetail in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142155.

WriteDebug

To provide information that can be used to troubleshoot a function, make the function call the WriteDebug method. This displays debug messages to the user. For more information, see

System.Management.Automation.Cmdlet.WriteDebug in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142156.

WriteError

Functions should call this method when nonterminating errors occur and the function is designed to continue processing records. For more information, see System.Management.Automation.Cmdlet.WriteError in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142157.

Note: If a terminating error occurs, the function should call the ThrowTerminatingError method.

WriteObject

This method allows the function to send an object to the next command in the pipeline. In most cases, this is the method to use when the function returns data. For more information, see System.Management.Automation.PSCmdlet.WriteObject in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142158.

WriteProgress

For functions whose actions take a long time to complete, this method allows the function to call the WriteProgress method so that progress information is displayed. For example, you can display the percent completed. For more information, see System.Management.Automation.PSCmdlet.WriteProgress in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142160.

WriteVerbose

To provide detailed information about what the function is doing, make the function call the WriteVerbose method to display verbose messages to the user. By default, verbose messages are not displayed. For more information, see System.Management.Automation.PSCmdlet.WriteVerbose in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142162.

WriteWarning

To provide information about conditions that may cause unexpected results, make the function call the WriteWarning method to display warning messages to the user. By default, warning messages are displayed. For more information, see System.Management.Automation.PSCmdlet.WriteWarning in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142164.

Note: You can also display warning messages by configuring the WarningPreference variable or by using the Verbose and Debug

command-line options.

Other Methods and Properties

For information about the other methods and properties that can be accessed through the $PSCmdlet variable, see System.Management.Automation.PSCmdlet in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142139.

For example, the ParameterSetName property allows you to see the parameter set that is being used. Parameter sets allow you to create a function that performs different tasks based on the parameters that are specified when the function is run.

SEE ALSO

about_Functions
about_Functions_Advanced
about_Functions_Advanced_Parameters
about_Functions_CmdletBindingAttribute
about_Functions_OutputTypeAttribute

TOPIC

about_Functions_Advanced_Parameters

SHORT DESCRIPTION

Explains how to add parameters to advanced functions.

LONG DESCRIPTION

You can add parameters to the advanced functions that you write, and use parameter attributes and arguments to limit the parameter values that function users submit with the parameter.

The parameters that you add to your function are available to users in addition to the common parameters that Windows PowerShell adds automatically to all cmdlets and advanced functions. For more information about the Windows PowerShell common parameters, see about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

Beginning in Windows PowerShell 3.0, you can use splatting with @Args to represent the parameters in a command. This technique is valid on simple and advanced functions. For more information, see about_Functions (http://go.microsoft.com/fwlink/?LinkID=113231) and about_Splatting (http://go.microsoft.com/fwlink/?LinkID=262720).

Static Parameters

Static parameters are parameters that are always available in the function. Most parameters in Windows PowerShell cmdlets and scripts are static parameters.

The following example shows the declaration of a ComputerName parameter that has the following characteristics:

- It is mandatory (required).
- It takes input from the pipeline.
- It takes an array of strings as input.

```
Param
 (
  [parameter(Mandatory=$true,
  ValueFromPipeline=$true)]
  [String[]]
  $ComputerName
 )
```

Attributes of Parameters

This section describes the attributes that you can add to function parameters.

All attributes are optional. However, if you omit the CmdletBinding attribute, then to be recognized as an advanced function, the function must include the Parameter attribute.

You can add one or multiple attributes in each parameter declaration. There is no limit to the number of attributes that you can add to a parameter declaration.

The Parameter Attribute

The Parameter attribute is used to declare the attributes of function parameters.

The Parameter attribute is optional, and you can omit it if none of the parameters of your functions need attributes, but to be recognized as an advanced function (rather than a simple function), a function must have either the CmdletBinding attribute or the Parameter attribute, or both.

The Parameter attribute has arguments that define the characteristics of the parameter, such as whether the parameter is mandatory or optional.

Use the following syntax to declare the Parameter attribute, an argument, and an argument value. The parentheses that enclose the argument and its value must follow "Parameter" with no intervening space.

```
Param
  (
    [parameter(Argument=value)]
    $ParameterName
  )
```

Use commas to separate arguments within the parentheses. Use the following syntax to declare two arguments of the Parameter attribute.

```
Param
  (
    [parameter(Argument1=value1,
            Argument2=value2)]

  )
```

If you use the Parameter attribute without arguments (as an alternative to using the CmdletBinding attribute), the parentheses that follow the attribute name are still required.

```
Param
  (
    [parameter()]
    $ParameterName
  )
```

Mandatory Argument

The Mandatory argument indicates that the parameter is required. If this argument is not specified, the parameter is an optional parameter.

The following example declares the ComputerName parameter. It uses the Mandatory argument to make the parameter mandatory.

```
Param
 (
   [parameter(Mandatory=$true)]
   [String[]]
   $ComputerName
 )
```

Position Argument

The Position argument determines whether the parameter name is required when the parameter is used in a command. When a parameter declaration includes the Position argument, the parameter name can be omitted and Windows PowerShell identifies the unnamed parameter value by its position (or order) in the list of unnamed parameter values in the command.

If the Position argument is not specified, the parameter name (or a parameter name alias or abbreviation) must precede the parameter value whenever the parameter is used in a command.

By default, all function parameters are positional. Windows PowerShell assigns position numbers to parameters in the order in which the parameters are declared in the function. To disable this feature, set the value of the PositionalBinding argument of the CmdletBinding attribute to $False. The Position argument takes precedence over the value of the PositionalBinding argument for the parameters on which it is declared. For more information, see PositionalBinding in about_Functions_CmdletBindingAttribute.

The value of the Position argument is specified as an integer. A position value of 0 represents the first position in the command, a position value of 1 represents the second position in the command, and so on.

If a function has no positional parameters, Windows PowerShell assigns positions to each parameter based on the order in which the parameters are declared. However, as a best practice, do not rely on this assignment. When you want parameters to be positional, use the Position argument.

The following example declares the ComputerName parameter. It uses the Position argument with a value of 0. As a result, when "-ComputerName" is omitted from command, its value must be the first or only unnamed parameter value in the command.

```
Param
 (
  [parameter(Position=0)]
  [String[]]
  $ComputerName
 )
```

NOTE: When the Get-Help cmdlet displays the corresponding "Position?" parameter attribute, the position value is incremented by 1. For example, a parameter with a Position argument value of 0 has a parameter attribute of "Position? 1."

ParameterSetName Argument

The ParameterSetName argument specifies the parameter set to which a parameter belongs. If no parameter set is specified, the parameter belongs to all the parameter sets defined by the function. Therefore, to be unique, each parameter set must have at least one parameter that is not a member of any other parameter set.

The following example declares a ComputerName parameter in the Computer parameter set, a UserName parameter in the User parameter set, and a Summary parameter in both parameter sets.

```
Param
 (
  [parameter(Mandatory=$true,
        ParameterSetName="Computer")]
  [String[]]
  $ComputerName,

  [parameter(Mandatory=$true,
        ParameterSetName="User")]
  [String[]]
  $UserName,

  [parameter(Mandatory=$false)]
  [Switch]
  $Summary
 )
```

You can specify only one ParameterSetName value in each argument and only one ParameterSetName argument in each Parameter attribute. To indicate that a parameter appears in more than one parameter set, add additional Parameter

attributes.

The following example explicitly adds the Summary parameter to the Computer and User parameter sets. The Summary parameter is mandatory in one parameter set and optional in the other.

```
Param
 (
  [parameter(Mandatory=$true,
        ParameterSetName="Computer")]
  [String[]]
  $ComputerName,

  [parameter(Mandatory=$true,
        ParameterSetName="User")]
  [String[]]
  $UserName,

  [parameter(Mandatory=$false, ParameterSetName="Computer")]
  [parameter(Mandatory=$true, ParameterSetName="User")]
  [Switch]
  $Summary
 )
```

For more information about parameter sets, see "Cmdlet Parameter Sets" in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142183.

ValueFromPipeline Argument

The ValueFromPipeline argument indicates that the parameter accepts input from a pipeline object. Specify this argument if the function accepts the entire object, not just a property of the object.

The following example declares a ComputerName parameter that is mandatory and accepts an object that is passed to the function from the pipeline.

```
Param
 (
  [parameter(Mandatory=$true,
        ValueFromPipeline=$true)]
  [String[]]
  $ComputerName
 )
```

ValueFromPipelineByPropertyName Argument

The valueFromPipelineByPropertyName argument indicates that the parameter accepts input from a property of a pipeline object. The object property must have the same name or alias as the parameter.

For example, if the function has a ComputerName parameter, and the piped object has a ComputerName property, the value of the ComputerName property is assigned to the ComputerName parameter of the function.

The following example declares a ComputerName parameter that is mandatory and accepts input from the ComputerName property of the object that is passed to the function through the pipeline.

```
Param
 (
  [parameter(Mandatory=$true,
        ValueFromPipelineByPropertyName=$true)]
  [String[]]
  $ComputerName
 )
```

ValueFromRemainingArguments Argument

The ValueFromRemainingArguments argument indicates that the parameter accepts all of the parameters values in the command that are not assigned to other parameters of the function.

The following example declares a ComputerName parameter that is mandatory and accepts all the remaining parameter values that were submitted to the function.

```
Param
 (
  [parameter(Mandatory=$true,
        ValueFromRemainingArguments=$true)]
  [String[]]
  $ComputerName
 )
```

HelpMessage Argument

The HelpMessage argument specifies a string that contains a brief description of the parameter or its value. Windows PowerShell displays this message in the prompt that appears when a mandatory parameter value is missing from a command. This argument has no effect on optional parameters.

The following example declares a mandatory ComputerName parameter and a help message that explains the expected parameter value.

```
Param
 (
  [parameter(mandatory=$true,
        HelpMessage="Enter one or more computer names separated by commas.")]
  [String[]]
  $ComputerName
 )
```

Alias Attribute

The Alias attribute establishes an alternate name for the parameter. There is no limit to the number of aliases that you can assign to a parameter.

The following example shows a parameter declaration that adds the "CN" and "MachineName" aliases to the mandatory ComputerName parameter.

```
Param
 (
  [parameter(Mandatory=$true)]
  [alias("CN","MachineName")]
  [String[]]
  $ComputerName
 )
```

Parameter and Variable Validation Attributes

Validation attributes direct Windows PowerShell to test the parameter values that users submit when they call the advanced function. If the parameter values fail the test, an error is generated and the function is not called. You can also use some of the validation attributes to restrict the values that users can specify for variables.

AllowNull Validation Attribute

The AllowNull attribute allows the value of a mandatory parameter to be null ($null). The following example declares a ComputerName parameter that can have a Null value.

```
Param
```

```
 (
  [parameter(Mandatory=$true)]
  [AllowNull()]
  [String]
  $ComputerName
 )
```

AllowEmptyString Validation Attribute

The AllowEmptyString attribute allows the value of a mandatory parameter to be an empty string (""). The following example declares a ComputerName parameter that can have an empty string value.

```
Param
 (
  [parameter(Mandatory=$true)]
  [AllowEmptyString()]
  [String]
  $ComputerName
 )
```

AllowEmptyCollection Validation Attribute

The AllowEmptyCollection attribute allows the value of a mandatory parameter to be an empty collection (@()). The following example declares a ComputerName parameter that can have a empty collection value.

```
Param
 (
  [parameter(Mandatory=$true)]
  [AllowEmptyCollection()]
  [String[]]
  $ComputerName
 )
```

ValidateCount Validation Attribute

The ValidateCount attribute specifies the minimum and maximum number of parameter values that a parameter accepts. Windows PowerShell generates an error if the number of parameter values in the command that calls the function is outside that range.

The following parameter declaration creates a ComputerName parameter that takes 1 to 5 parameter values.

```
Param
 (
  [parameter(Mandatory=$true)]
  [ValidateCount(1,5)]
  [String[]]
  $ComputerName
 )
```

ValidateLength Validation Attribute

The ValidateLength attribute specifies the minimum and maximum number of characters in a parameter or variable value. Windows PowerShell generates an error if the length of a value specified for a parameter or a variable is outside of the range.

In the following example, each computer name must have one to 10 characters.

```
Param
 (
  [parameter(Mandatory=$true)]
  [ValidateLength(1,10)]
  [String[]]
  $ComputerName
 )
```

In the following example, the value of the variable $number must be a minimum of one character in length, and a maximum of ten characters.

```
[Int32][ValidateLength(1,10)]$number = 01
```

ValidatePattern Validation Attribute

The ValidatePattern attribute specifies a regular expression that is compared to the parameter or variable value. Windows PowerShell generates an error if the value does not match the regular expression pattern.

In the following example, the parameter value must be a four-digit number, and each digit must be a number 0 to 9.

```
Param
 (
  [parameter(Mandatory=$true)]
  [ValidatePattern("[0-9][0-9][0-9][0-9]")]
  [String[]]
```

```
  $ComputerName
 )
```

In the following example, the value of the variable $number must be a four-digit number, and each digit must be a number 0 to 9.

```
[Int32][ValidatePattern("[0-9][0-9][0-9][0-9]")]$number = 1111
```

ValidateRange Validation Attribute

The ValidateRange attribute specifies a numeric range for each parameter or variable value. Windows PowerShell generates an error if any value is outside that range. In the following example, the value of the Attempts parameter must be between 0 and 10.

```
Param
 (
  [parameter(Mandatory=$true)]
  [ValidateRange(0,10)]
  [Int]
  $Attempts
 )
```

In the following example, the value of the variable $number must be between 0 and 10.

```
[Int32][ValidateRange(0,10)]$number = 5
```

ValidateScript Validation Attribute

The ValidateScript attribute specifies a script that is used to validate a parameter or variable value. Windows 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 is being validated is mapped to the $_ variable. You can use the $_ variable to refer to the value in the script.

In the following example, the value of the EventDate parameter must be greater than or equal to the current date.

```
Param
 (
  [parameter()]
  [ValidateScript({$_ -ge (get-date)})]
  [DateTime]
```

```
  $EventDate
 )
```

In the following example, the value of the variable $date must be greater than or equal to the current date and time.

```
[DateTime][ValidateScript({$_ -ge (get-date)})]$date = (get-date)
```

ValidateSet Attribute

The ValidateSet attribute specifies a set of valid values for a parameter or variable. Windows PowerShell generates an error if a parameter or variable value does not match a value in the set. In the following example, the value of the Detail parameter can only be "Low," "Average," or "High."

```
Param
 (
  [parameter(Mandatory=$true)]
  [ValidateSet("Low", "Average", "High")]
  [String[]]
  $Detail
 )
```

In the following example, the value of the variable $flavor must be either Chocolate, Strawberry, or Vanilla.

```
[String][ValidateSet("Chocolate", "Strawberry", "Vanilla")]$flavor = Strawberry
```

ValidateNotNull Validation Attribute

The ValidateNotNull attribute specifies that the parameter value cannot be null ($null). Windows PowerShell generates an error if the parameter value is null.

The ValidateNotNull attribute is designed to be used when the type of the parameter value is not specified or when the specified type will accept a value of Null. (If you specify a type that will not accept a null value, such as a string, the null value will be rejected without the ValidateNotNull attribute, because it does not match the specified type.)

In the following example, the value of the ID parameter cannot be null.

```
Param
 (
  [parameter(Mandatory=$true)]
  [ValidateNotNull()]
  $ID
 )
```

ValidateNotNullOrEmpty Validation Attribute

The ValidateNotNullOrEmpty attribute specifies that the parameter value cannot be null ($null) and cannot be an empty string (""). Windows PowerShell generates an error if the parameter is used in a function call, but its value is null, an empty string, or an empty array.

```
Param
 (
  [parameter(Mandatory=$true)]
  [ValidateNotNullOrEmpty()]
  [String[]]
  $UserName
 )
```

Dynamic Parameters

Dynamic parameters are parameters of a cmdlet, function, or script that are available only under certain conditions.

For example, several provider cmdlets have parameters that are available only when the cmdlet is used in the provider drive, or in a particular path of the provider drive. For example, the Encoding parameter is available on the Add-Content, Get-Content, and Set-Content cmdlets only when it is used in a file system drive.

You can also create a parameter that appears only when another parameter is used in the function command or when another parameter has a certain value.

Dynamic parameters can be very useful, but use them only when necessary, because they can be difficult for users to discover. To find a dynamic parameter, the user must be in the provider path, use the ArgumentList parameter of the Get-Command cmdlet, or use the Path parameter of Get-Help.

To create a dynamic parameter for a function or script, use the DynamicParam keyword.

The syntax is as follows:

DynamicParam {<statement-list>}

In the statement list, use an If statement to specify the conditions under which the parameter is available in the function.

Use the New-Object cmdlet to create a System.Management.Automation.RuntimeDefinedParameter object to represent the parameter and specify its name.

You can also use a New-Object command to create a System.Management.Automation.ParameterAttribute object to represent attributes of the parameter, such as Mandatory, Position, or ValueFromPipeline or its parameter set.

The following example shows a sample function with standard parameters named 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 Get-Sample {
   [CmdletBinding()]
   Param ([String]$Name, [String]$Path)

   DynamicParam
   {
      if ($path -match ".*HKLM.*:")
      {
         $attributes = new-object System.Management.Automation.ParameterAttribute
         $attributes.ParameterSetName = "__AllParameterSets"
         $attributes.Mandatory = $false
         $attributeCollection = new-object `
            -Type System.Collections.ObjectModel.Collection[System.Attribute]
         $attributeCollection.Add($attributes)

         $dynParam1 = new-object `
            -Type System.Management.Automation.RuntimeDefinedParameter("dp1", [Int32],
$attributeCollection)

         $paramDictionary = new-object `
            -Type System.Management.Automation.RuntimeDefinedParameterDictionary
         $paramDictionary.Add("dp1", $dynParam1)
         return $paramDictionary
```

```
        }
    }
}
```

For more information, see "RuntimeDefinedParameter Class" in
the MSDN (Microsoft Developer Network) library at
http://go.microsoft.com/fwlink/?LinkID=145130.

Switch Parameters

Switch parameters are parameters with no parameter value. They
are effective only when they are used and have only one effect.

For example, the -NoProfile parameter of PowerShell.exe is
a switch parameter.

To create a switch parameter in a function, specify the Switch type
in the parameter definition.

For example:

```
    Param ([Switch]<ParameterName>)
-or-
    Param
     (
       [parameter(Mandatory=$false)]
       [Switch]
       $<ParameterName>
     )
```

Switch parameters are easy to use and are preferred over Boolean
parameters, which have a more difficult syntax.

For example, to use a switch parameter, the user types the parameter
in the command.

```
    -IncludeAll
```

To use a Boolean parameter, the user types the parameter and a Boolean
value.

```
    -IncludeAll:$true
```

When creating switch parameters, choose the parameter name carefully.
Be sure that the parameter name communicates the effect of the parameter
to the user, and avoid ambiguous terms, such as Filter or Maximum, that
might imply that a value is required.

SEE ALSO

about_Functions
about_Functions_Advanced
about_Functions_Advanced_Methods
about_Functions_CmdletBindingAttribute
about_Functions_OutputTypeAttribute

TOPIC

about_Functions_CmdletBindingAttribute

SHORT DESCRIPTION

Describes the attribute that makes a function work like a compiled cmdlet.

LONG DESCRIPTION

The CmdletBinding attribute is an attribute of functions that makes them operate like compiled cmdlets that are written in C#, and it provides access to features of cmdlets.

Windows PowerShell binds the parameters of functions that have the CmdletBinding attribute in the same way that it binds the parameters of compiled cmdlets. The $PSCmdlet automatic variable is available to functions with the CmdletBinding attribute, but the $Args variable is not available.

In functions that have the CmdletBinding attribute, unknown parameters and positional arguments that have no matching positional parameters cause parameter binding to fail.

Note: Compiled cmdlets use the required Cmdlet attribute, which is similar to the CmdletBinding attribute that is described in this topic.

SYNTAX

The following example shows the format of a function that specifies all the optional arguments of the CmdletBinding attribute. A brief description of each argument follows this example.

```
{
 [CmdletBinding(ConfirmImpact=<String>,
        DefaultParameterSetName=<String>,
        HelpURI=<URI>,
        SupportsPaging=<Boolean>,
        SupportsShouldProcess=<Boolean>,
        PositionalBinding=<Boolean>)]

 Param ($Parameter1)
 Begin{}
 Process{}
 End{}
}
```

ConfirmImpact

The ConfirmImpact argument specifies when the action of the function should be confirmed by a call to the ShouldProcess method. The call to the ShouldProcess method displays a confirmation prompt only when the ConfirmImpact argument is equal to or greater than the value of the $ConfirmPreference preference variable. (The default value of the argument is Medium.) Specify this argument only when the SupportsShouldProcess argument is also specified.

For more information about confirmation requests, see "Requesting Confirmation" in the MSDN (Microsoft Developer Network) library at http://go.microsoft.com/fwlink/?LinkId=136658.

DefaultParameterSetName

The DefaultParameterSetName argument specifies the name of the parameter set that Windows PowerShell will attempt to use when it cannot determine which parameter set to use. You can avoid this issue by making the unique parameter of each parameter set a mandatory parameter.

HelpURI

The HelpURI argument specifies the Internet address (Uniform Resource Identifier (URI)) of the online version of the help topic that describes the function. The value of the HelpURI argument must begin with "http" or "https".

The HelpURI argument value is used for the value of the HelpURI property

of the CommandInfo object that Get-Command returns for the function.

However, when help files are installed on the computer and the value of the first link in the RelatedLinks section of the help file is a URI, or the value of the first .Link directive in comment-based help is a URI, the URI in the help file is used as the value of the HelpUri property of the function.

The Get-Help cmdlet uses the value of the HelpURI property to locate the online version of the function help topic when the Online parameter of Get-Help is specified in a command.

SupportsPaging

The SupportsPaging argument adds the First, Skip, and IncludeTotalCount parameters to the function. These parameters allow users to select output from a very large result set. This argument is designed for cmdlets and functions that return data from large data stores that support data selection, such as a SQL database.

This argument was introduced in Windows PowerShell 3.0.

-- First: Gets only the first 'n' objects.
-- Skip: Ignores the first 'n' objects and then gets the remaining objects.
-- IncludeTotalCount: Reports the number of objects in the data set (an integer) followed by the objects. If the cmdlet cannot determine the total count, it returns "Unknown total count".

Windows PowerShell includes NewTotalCount, a helper method that gets the total count value to return and includes an estimate of the accuracy of the total count value.

The following sample function shows how to add support for the paging parameters to an advanced function.

```
function Get-Numbers
{
   [CmdletBinding(SupportsPaging = $true)]
   param()

   $FirstNumber = [Math]::Min($PSCmdlet.PagingParameters.Skip, 100)
   $LastNumber = [Math]::Min($PSCmdlet.PagingParameters.First + $FirstNumber - 1, 100)

   if ($PSCmdlet.PagingParameters.IncludeTotalCount)
   {
      $TotalCountAccuracy = 1.0
      $TotalCount = $PSCmdlet.PagingParameters.NewTotalCount(100, $TotalCountAccuracy)
      Write-Output $TotalCount
```

```
    }
    $FirstNumber .. $LastNumber | Write-Output
  }
```

SupportsShouldProcess

The SupportsShouldProcess argument adds Confirm and WhatIf parameters to the function. The Confirm parameter prompts the user before it runs the command on each object in the pipeline. The WhatIf parameter lists the changes that the command would make, instead of running the command.

PositionalBinding

The PositionalBinding argument determines whether parameters in the function are positional by default. The default value is $True. You can use the PositionalBinding argument with a value of $False to disable positional binding.

The PositionalBinding argument is introduced in Windows PowerShell 3.0.

When parameters are positional, the parameter name is optional. Windows PowerShell associates unnamed parameter values with the function parameters according to the order or position of the unnamed parameter values in the function command.

When parameters are not positional (they are "named"), the parameter name (or an abbreviation or alias of the name) is required in the command.

When PositionalBinding is $True, function parameters are positional by default. Windows PowerShell assigns position number to the parameters in the order in which they are declared in the function.

When PositionalBinding is $False, function parameters are not positional by default. Unless the Position argument of the Parameter attribute is declared on the parameter, the parameter name (or an alias or abbreviation) must be included when the parameter is used in a function.

The Position argument of the Parameter attribute takes precedence over the PositionalBinding default value. You can use the Position argument to specify a position value for a parameter. For more information about the Position argument, see about_Functions_Advanced_Parameters (http://go.microsoft.com/fwlink/?LinkID=135173).

NOTES

The SupportsTransactions argument is not supported in advanced functions.

KEYWORDS

about_Functions_CmdletBinding_Attribute

SEE ALSO

about_Functions
about_Functions_Advanced
about_Functions_Advanced_Methods
about_Functions_Advanced_Parameters
about_Functions_OutputTypeAttribute

TOPIC

about_Functions_OutputTypeAttribute

SHORT DESCRIPTION

Describes an attribute that reports the type of object that the function returns.

LONG DESCRIPTION

The OutputType attribute lists the .NET types of objects that the functions returns. You can use its optional ParameterSetName parameter to list different output types for each parameter set.

The OutputType attribute is supported on simple and advanced functions. It is independent of the CmdletBinding attribute.

The OutputType attribute provides the value of the OutputType property of the System.Management.Automation.FunctionInfo object that the Get-Command cmdlet returns.

The OutputType attribute value is only a documentation note. It is not derived from the function code or compared to the actual function output. As such, the value might be inaccurate.

SYNTAX

The OutputType attribute of functions has the following syntax:

```
[OutputType([<TypeLiteral>], ParameterSetName="<Name>")]
[OutputType("<TypeNameString>", ParameterSetName="<Name>")]
```

The ParameterSetName parameter is optional.

You can list multiple types in the OutputType attribute.

```
[OutputType([<Type1>],[<Type2>],[<Type3>])]
```

You can use the ParameterSetName parameter to indicate that different parameter sets return different types.

```
[OutputType([<Type1>], ParameterSetName="<Set1>","<Set2>")]
[OutputType([<Type2>], ParameterSetName="<Set3>")]
```

Place the OutputType attribute statements in the attributes list that precedes the Param statement.

The following example shows the placement of the OutputType attribute in a simple function.

```
function SimpleFunction2
{
[OutputType([<Type>])]
    Param ($Parameter1)

    <function body>
}
```

The following example shows the placement of the OutputType attribute in advanced functions.

```
function AdvancedFunction1
{
  [OutputType([<Type>])]
  Param (
    [parameter(Mandatory=$true)]
    [String[]]
    $Parameter1
  )

 <function body>
```

```
}


function AdvancedFunction2
{
   [CmdletBinding(SupportsShouldProcess=<Boolean>)]
   [OutputType([<Type>])]
   Param (
      [parameter(Mandatory=$true)]
      [String[]]
      $Parameter1
   )

  <function body>
}
```

EXAMPLES

The following function uses the OutputType attribute to indicate that it returns a string value.

```
function Send-Greeting
{
  [OutputType([String])]
  Param ($Name)

  Hello, $Name
}
```

To see the resulting output type property, use the Get-Command cmdlet.

```
PS C:\> (Get-Command Send-Greeting).OutputType

Name                                Type
----                               ----
System.String                           System.String
```

The following advanced function uses the OutputType attribute to indicate that the function returns different types depending on the parameter set used in the function command.

```
function Get-User
{
   [CmdletBinding(DefaultParameterSetName="ID")]
```

```
    [OutputType("System.Int32", ParameterSetName="ID")]
    [OutputType([String], ParameterSetName="Name")]

    Param (
      [parameter(Mandatory=$true, ParameterSetName="ID")]
      [Int[]]
      $UserID,

      [parameter(Mandatory=$true, ParameterSetName="Name")]
      [String[]]
      $UserName
    )

   <function body>
  }
```

The following example demonstrates that the output type property value displays the value of the OutputType attribute, even when it is inaccurate.

The Get-Time function returns a string that contains the short form of the time in any DateTime object. However, the OutputType attribute reports that it returns a System.DateTime object.

```
  function Get-Time
  {
    [OutputType([DateTime])]
    Param
    (
      [parameter(Mandatory=$true)]
      [Datetime]$DateTime
    )
    $DateTime.ToShortTimeString()
  }
```

The Get-Type method confirms that the function returns a string.

```
  PS C:\> (Get-Time -DateTime (Get-Date)).Gettype().FullName
  System.String
```

However, the OutputType property, which gets its value from the OutputType attribute, reports that the function returns a DateTime object.

```
  PS C:\> (Get-Command Get-Time).OutputType
```

```
Name                           Type
----                           ----
System.DateTime                System.DateTime
```

NOTES

The value of the OutputType property of a FunctionInfo object is an array of System.Management.Automation.PSTypeName objects, each of which have Name and Type properties.

To get only the name of each output type, use a command with the following format.

```
(Get-Command Get-Time).OutputType | ForEach {$_.Name}
```

The value of the OutputType property can be null. Use a null value when the output is a not a .NET type, such as a WMI object or a formatted view of an object.

SEE ALSO

about_Functions
about_Functions_Advanced
about_Functions_Advanced_Methods
about_Functions_Advanced_Parameters
about_Functions_CmdletBindingAttribute

TOPIC

about_Group_Policy_Settings

SHORT DESCRIPTION

Describes the Group Policy settings for Windows PowerShell

LONG DESCRIPTION

Windows PowerShell includes Group Policy settings to help you define consistent option values for servers in an enterprise environment.

The Windows PowerShell Group Policy settings are in the following Group Policy paths:

Computer Configuration\Administrative Templates\Windows Components\Windows PowerShell
User Configuration\Administrative Templates\Windows Components\Windows PowerShell

Group policy settings in the User Configuration path take precedence over Group Policy settings in the Computer Configuration path.

The policies are as follows:

Turn on Script Execution: Sets the Windows PowerShell execution policy.
Turn on Module Logging: Sets the LogPipelineExecutionDetails property of modules.
Set the default source path for Update-Help: Sets the source for UpdatableHelp to a directory, not the Internet.

To download spreadsheets that list all of the Group Policy settings for each version of Windows, see "Group Policy Settings Reference for Windows and Windows Server" in the Microsoft Download Center at http://go.microsoft.com/fwlink/?LinkId=261775

TURN ON SCRIPT EXECUTION

The "Turn on Script Execution" policy setting sets the execution policy for computers and users, which determines which scripts are permitted to run.

If you enable the policy setting, you can select from among the following policy settings.

-- "Allow only signed scripts" allows scripts to execute only if they are signed by a trusted publisher. This policy setting is equivalent to the AllSigned execution policy.

-- "Allow local scripts and remote signed scripts" allows all local scripts to run. Scripts that originate from the Internet must be signed by a trusted publisher. This policy setting is equivalent to the RemoteSigned execution policy.

-- "Allow all scripts" allows all scripts to run. This policy setting is equivalent to the Unrestricted execution policy.

If you disable this policy setting, no scripts are allowed to run.
This policy setting is equivalent to the Restricted execution policy.

If you disable or do not configure this policy setting, the execution policy that is set for the computer or user by the Set-ExecutionPolicy cmdlet determines whether scripts are permitted to run. The default value is Restricted.

For more information, see about_Execution_Policies (http://go.microsoft.com/fwlink/?LinkID=135170).

TURN ON MODULE LOGGING

The "Turn on Module Logging" policy setting turns on logging for selected Windows PowerShell modules. The setting is effective in all sessions on all affected computers.

If you enable this policy setting and specify one or more modules, pipeline execution events for the specified modules are recorded in the Windows PowerShell log in Event Viewer.

If you disable this policy setting, logging of execution events is disabled for all Windows PowerShell modules.

If this policy setting is not configured, the LogPipelineExecutionDetails property of each module or snap-in determines whether the execution events of a module or snap-in are logged. By default, the LogPipelineExecutionDetails property of all modules and snap-ins is set to False.

To turn on module logging for a module, use the following command format. The module must be imported into the session and the setting is effective only in the current session.

```
PS C:\>Import-Module <Module-Name>
PS C:\>(Get-Module <Module-Name>).LogPipelineExecutionDetails = $true
```

To turn on module logging for all sessions on a particular computer, add the previous commands to the all-users Windows PowerShell profile ($Profile.AllUsers.AllHosts).

For more information about module logging, see about_Modules (http://go.microsoft.com/fwlink/?LinkID=144311).

SET THE DEFAULT SOURCE PATH FOR UPDATE-HELP

The "Set the Default Source Path for Update-Help" policy setting sets a default value for the SourcePath parameter of the Update-Help cmdlet. This setting prevents users from using the Update-Help cmdlet to download help files from the Internet.

NOTE: The "Set the default source path for Update-Help" Group Policy setting appears under Computer Configuration and User Configuration. However, only the Group Policy setting under Computer Configuration is effective. The Group Policy setting under User Configuration is ignored.

The Update-Help cmdlet downloads and installs the newest help files for Windows PowerShell modules and installs them on the computer. By default, Update-Help downloads new help files from an Internet location specified by the module.

However, you can use the Save-Help cmdlet to download the newest help files to a file system location, such as a network share, and then use the Update-Help cmdlet to get the help files from the file system location and install them on the computer. The SourcePath parameter of the Update-Help cmdlet specifies the file system location.

By providing a default value for the SourcePath parameter, this Group Policy setting implicitly adds the SourcePath parameter to all Update-Help commands. Users can override the particular file system location specified as the default value by entering a different file system location. But they cannot remove the SourcePath parameter from the Update-Help command.

If you enable this policy setting, you can specify a default value for the SourcePath parameter. Enter a file system location.

If this policy setting is disabled or not configured, there is no default value for the SourcePath parameter of the Update-Help cmdlet. Users can download help from the Internet or from any file system location.

For more information, see about_Updatable_Help (http://go.microsoft.com/fwlink/?LinkID=235801).

KEYWORDS

about_Group_Policies
about_GroupPolicy

SEE ALSO

about_Execution_Policies
about_Modules
about_Updatable_Help

Get-ExecutionPolicy
Set-ExecutionPolicy
Get-Module
Update-Help
Save-Help

TOPIC

about_Hash_Tables

SHORT DESCRIPTION

Describes how to create, use, and sort hash tables in Windows PowerShell.

LONG DESCRIPTION

A hash table, also known as a dictionary or associative array, is a compact data structure that stores one or more key/value pairs. For example, a hash table might contain a series of IP addresses and computer names, where the IP addresses are the keys and the computer names are the values, or vice versa.

In Windows PowerShell, each hash table is a Hashtable (System.Collections.Hashtable) object. You can use the properties and methods of Hashtable objects in Windows PowerShell.

Beginning in Windows PowerShell 3.0, you can use the [ordered] attribute to create an ordered dictionary (System.Collections.Specialized.OrderedDictionary) in Windows PowerShell.

Ordered dictionaries differ from hash tables in that the keys always appear in the order in which you list them. The order of keys in a hash table is not determined.

The keys and value in hash tables are also .NET objects. They are most often strings or integers, but they can have any object type. You can also create nested hash tables, in which the value of a key is another hash table.

Hash tables are frequently used because they are very efficient for finding and retrieving data. You can use hash tables to store lists and to create calculated properties in Windows PowerShell. And, Windows PowerShell has a cmdlet, ConvertFrom-StringData, that converts strings to a hash table.

Syntax

The syntax of a hash table is as follows:

```
@{ <name> = <value>; [<name> = <value> ] ...}
```

The syntax of an ordered dictionary is as follows:

```
[ordered]@{ <name> = <value>; [<name> = <value> ] ...}
```

The [ordered] attribute was introduced in Windows PowerShell 3.0.

Creating Hash Tables

To create a hash table, follow these guidelines:

- Begin the hash table with an at sign (@).
- Enclose the hash table in braces ({}).
- Enter one or more key/value pairs for the content of the hash table.
- Use an equal sign (=) to separate each key from its value.
- Use a semicolon (;) or a line break to separate the key/value pairs.
- Key that contains spaces must be enclosed in quotation marks. Values must be valid Windows PowerShell expressions. Strings must appear in quotation marks, even if they do not include spaces.
- To manage the hash table, save it in a variable.
- When assigning an ordered hash table to a variable, place the [ordered] attribute before the "@" symbol. If you place it before the variable name, the command fails.

To create an empty hash table in the value of $hash, type:

```
$hash = @{}
```

You can also add keys and values to a hash table when you create it. For example, the following statement creates a hash table with three keys.

```
$hash = @{ Number = 1; Shape = "Square"; Color = "Blue"}
```

Creating Ordered Dictionaries

You can create an ordered dictionary by adding an object of the OrderedDictiory type, but the easiest way to create an ordered dictionary is use the [Ordered] attribute.

The [ordered] attribute is introduced in Windows PowerShell 3.0.

Place the attribute immediately before the "@" symbol.

```
$hash = [ordered]@{ Number = 1; Shape = "Square"; Color = "Blue"}
```

You can use ordered dictionaries in the same way that you use hash tables. Either type can be used as the value of parameters that take a hash table or dictionary (iDictionary).

You cannot use the [ordered] attribute to convert or cast a hash hash table. If you place the ordered attribute before the variable name, the command fails with the following error message.

```
PS C:\> [ordered]$hash = @{}
At line:1 char:1
+ [ordered]$hash = @{}
+ ~~~~~~~~~~~~~~~
The ordered attribute can be specified only on a hash literal node.
+ CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
+ FullyQualifiedErrorId : OrderedAttributeOnlyOnHashLiteralNode
```

To correct the expression, move the [ordered] attribute.

```
PS C:\> $hash = [ordered]@{}
```

You can cast an ordered dictionary to a hash table, but you cannot recover the ordered attribute, even if you clear the variable and enter new values. To re-establish the order, you must remove and

recreate the variable.

```
PS C:\> [hashtable]$hash = [ordered]@{ Number = 1; Shape = "Square"; Color = "Blue"}
PS C:\ps-test> $hash

Name                   Value
----                   -----
Color                  Blue
Shape                  Square
Number                 1
```

Displaying Hash Tables

To display a hash table that is saved in a variable, type the variable name. By default, a hash tables is displayed as a table with one column for keys and one for values.

```
C:\PS> $hash

Name                   Value
----                   -----
Shape                  Square
Color                  Blue
Number                 1
```

Hash tables have Keys and Values properties. Use dot notation to display all of the keys or all of the values.

```
C:\PS> $hash.keys
Number
Shape
Color

C:\PS> $hash.values
1
Square
Blue
```

Each key name is also a property of the hash table, and its value is the value of the key-name property. Use the following format to display the property values.

```
$hashtable.<key>
<value>
```

For example:

```
C:\PS> $hash.Number
1

C:\PS> $hash.Color
Blue
```

Hash tables have a Count property that indicates the number of key-value pairs in the hash table.

```
C:\PS> $hash.count
3
```

Hash table tables are not arrays, so you cannot use an integer as an index into the hash table, but you can use a key name to index into the hash table. If the key is a string value, enclose the key name in quotation marks.

For example:

```
C:\PS> $hash["Number"]
1
```

Adding and Removing Keys and Values

To add keys and values to a hash table, use the following command format.

```
$hash["<key>"] = "<value>"
```

For example, to add a "Time" key with a value of "Now" to the hash table, use the following statement format.

```
$hash["Time"] = "Now"
```

You can also add keys and values to a hash table by using the Add method of the System.Collections.Hashtable object. The Add method has the following syntax:

```
Add(Key, Value)
```

For example, to add a "Time" key with a value of "Now" to the hash table, use the following statement format.

```
$hash = $hash.Add("Time", "Now")
```

And, you can add keys and values to a hash table by using the addition operator (+) to add a hash table to an existing hash table. For example, the following statement adds a "Time" key with a value of "Now" to the hash table in the $hash variable.

```
$hash = $hash + @{Time="Now"}
```

You can also add values that are stored in variables.

```
$t = "Today"
$now = (Get-Date)

$hash.Add($t, $now)
```

You cannot use a subtraction operator to remove a key/value pair from a hash table, but you can use the Remove method of the Hashtable object. The Remove method takes the key as its value.

The Remove method has the following syntax:

```
Remove(Key)
```

For example, to remove the Time=Now key/value pair from the hash table in the value of the $hash variable, type:

```
$hash.$Remove("Time")
```

You can use all of the properties and methods of Hashtable objects in Windows PowerShell, including Contains, Clear, Clone, and CopyTo. For more information about Hashtable objects, see "System.Collections.Hashtable" on MSDN.

Object Types in HashTables

The keys and values in a hash table can have any .NET object type, and a single hash table can have keys and values of multiple types.

The following statement creates a hash table of process name strings and process object values and saves it in the $p variable.

```
$p = @{"PowerShell" = (get-process PowerShell);
"Notepad" = (get-process notepad)}
```

You can display the hash table in $p and use the key-name properties to display the values.

```
C:\PS> $p

Name                   Value
----                   -----
PowerShell             System.Diagnostics.Process (PowerShell)
Notepad                System.Diagnostics.Process (notepad)



C:\PS> $p.PowerShell

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    441     24    54196      54012   571    5.10   1788 PowerShell


C:\PS> $p.keys | foreach {$p.$_.handles}
441
251
```

The keys in a hash table can also be any .NET type. The following statement adds a key/value pair to the hash table in the $p variable. The key is a Service object that represents the WinRM service, and the value is the current status of the service.

```
C:\PS> $p = $p + @{(Get-Service WinRM) = ((Get-Service WinRM).Status)}
```

You can display and access the new key/value pair by using the same methods that you use for other pairs in the hash table.

```
C:\PS> $p

Name                   Value
----                   -----
PowerShell             System.Diagnostics.Process (PowerShell)
Notepad                System.Diagnostics.Process (notepad)
System.ServiceProcess.Servi... Running
```

```
C:\PS> $p.keys
PowerShell
Notepad

Status   Name               DisplayName
------   ----               -----------
Running  winrm              Windows Remote Management (WS-Manag...


C:\PS> $p.keys | foreach {$_.name}
winrm
```

The keys and values in a hash table can also be Hashtable objects. The following statement adds key/value pair to the hash table in the $p variable in which the key is a string, Hash2, and the value is a hash table with three key/value pairs.

```
C:\PS> $p = $p + @{"Hash2"= @{a=1; b=2; c=3}}
```

You can display and access the new values by using the same methods.

```
C:\PS> $p


Name                           Value
----                           -----
PowerShell                     System.Diagnostics.Process (PowerShell)
Notepad                        System.Diagnostics.Process (notepad)
System.ServiceProcess.Servi... Running
Hash2                          {a, b, c}


C:\PS> $p.Hash2

Name                           Value
----                           -----
a                              1
b                              2
c                              3


C:\PS> $p.Hash2.b
2
```

Sorting Keys and Values

The items in a hash table are intrinsically unordered. The key/value pairs might appear in a different order each time that you display them.

Although you cannot sort a hash table, you can use the GetEnumerator method of hash tables to enumerate the keys and values, and then use the Sort-Object cmdlet to sort the enumerated values for display.

For example, the following commands enumerate the keys and values in the hash table in the $p variable and then sort the keys in alphabetical order.

```
C:\PS> $p.GetEnumerator() | Sort-Object -Property key

Name                   Value
----                   -----
Notepad                System.Diagnostics.Process (notepad)
PowerShell             System.Diagnostics.Process (PowerShell)
System.ServiceProcess.Servi... Running
```

The following command uses the same procedure to sort the hash values in descending order.

```
C:\PS> $p.getenumerator() | Sort-Object -Property Value -Descending

Name                   Value
----                   -----
PowerShell             System.Diagnostics.Process (PowerShell)
Notepad                System.Diagnostics.Process (notepad)
System.ServiceProcess.Servi... Running
```

Creating Objects from Hash Tables

Beginning in Windows PowerShell 3.0, you can create an object from a hash table of properties and property values.

The syntax is as follows:

```
[<class-name>]@{<property-name>=<property-value>;<property-name>=<property-value>}
```

This method works only for classes that have a null constructor, that is, a constructor that has no parameters. The object properties must be public and settable.

For more information, see about_Object_Creation.

ConvertFrom-StringData

The ConvertFrom-StringData cmdlet converts a string or a here-string of key/value pairs into a hash table. You can use the ConvertFrom-StringData cmdlet safely in the Data section of a script, and you can use it with the Import-LocalizedData cmdlet to display user messages in the user-interface (UI) culture of the current user.

Here-strings are especially useful when the values in the hash table include quotation marks. (For more information about here-strings, see about_Quoting_Rules.)

The following example shows how to create a here-string of the user messages in the previous example and how to use ConvertFrom-StringData to convert them from a string into a hash table.

The following command creates a here-string of the key/value pairs and then saves it in the $string variable.

```
C:\PS> $string = @"
Msg1 = Type "Windows".
Msg2 = She said, "Hello, World."
Msg3 = Enter an alias (or "nickname").
"@
```

This command uses the ConvertFrom-StringData cmdlet to convert the here-string into a hash table.

```
C:\PS> ConvertFrom-StringData $string

Name                 Value
----                 -----
Msg3                 Enter an alias (or "nickname").
Msg2                 She said, "Hello, World."
Msg1                 Type "Windows".
```

For more information about here-strings, see about_Quoting_Rules.

SEE ALSO
about_Arrays
about_Object_Creation
about_Quoting_Rules
about_Script_Internationalization
ConvertFrom-StringData
Import-LocalizedData
"System.Collections.Hashtable" on MSDN

TOPIC

about_Hidden

SHORT DESCRIPTION

Describes the Hidden keyword, which hides class members from default Get-Member results.

LONG DESCRIPTION

When you use the Hidden keyword in a script, you hide the members of a class by default. The Hidden keyword can hide properties, methods (including constructors, events, alias properties, and other member types, including static members, from the default results of the Get-Member cmdlet, and from IntelliSense and tab completion results. To display members that you have hidden with the Hidden keyword, add the -Force parameter to a Get-Member command.

Hidden members are not displayed by using tab completion or IntelliSense, unless the completion occurs in the class that defines the hidden member.

A new attribute, System.Management.Automation.HiddenAttribute, has been added, so that C# code can have the same semantics within Windows PowerShell.

The Hidden keyword is useful for creating properties and methods within a class that you do not necessarily want other users of the class to see, or readily be able to edit.

The Hidden keyword has no effect on how you can view or make changes to members of a class. Like all language keywords in Windows PowerShell, Hidden is not case-sensitive, and hidden members are still public.

Hidden, along with custom classes, was introduced in Windows PowerShell 5.0.

EXAMPLE

The following example shows how to use the Hidden keyword in a class definition. The Car class method, Drive, has a property, rides, that does not need to be viewed or changed (it merely tallies the number of times that Drive is called on the Car class, a metric that is not important to users of the class; consider, for example, that when you are buying a car, you do not ask the seller on how many drives the car has been taken).

Because there is little need for users of the class to change this property, we can hide the property from Get-Member and automatic completion results by using the Hidden keyword.

Add the Hidden keyword by entering it on the same statement line as the property and its data type. Although the keyword can be in any order on this line, starting the statement with the Hidden keyword makes it easier for you later to identify all members that you have hidden.

```
class Car
{
  # Properties
  [String] $Color
  [String] $ModelYear
  [int] $Distance

  # Method
  [int] Drive ([int]$miles)
  {
    $this.Distance += $miles
    $this.rides++
    return $this.Distance
  }

  # Hidden property of the Drive method
   hidden [int] $rides = 0
}
```

Now, create a new instance of the Car class, and save it in a variable, $TestCar.

```
$TestCar = [Car]::new()
```

After you create the new instance, pipe the contents of the $TestCar variable to Get-Member. Observe that the rides property is not among the members listed in the Get-Member command results.

```
PS C:\Windows\system32> $TestCar | Get-Member

   TypeName: Car

Name        MemberType Definition
----        ---------- ----------
Drive       Method     int Drive(int miles)
Equals      Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
ToString    Method     string ToString()
Color       Property   string Color {get;set;}
Distance    Property   int Distance {get;set;}
ModelYear   Property   string ModelYear {get;set;}
```

Now, try running Get-Member again, but this time, add the -Force parameter. Note that the results contain the hidden rides property, among other members that are hidden by default.

```
PS C:\Windows\system32> $TestCar | Get-Member -Force

   TypeName: Car

Name         MemberType   Definition
----         ----------   ----------
pstypenames   CodeProperty System.Collections.ObjectModel.Collection`1…
psadapted     MemberSet    psadapted {Color, ModelYear, Distance,
psbase        MemberSet    psbase {Color, ModelYear, Distance,...
psextended    MemberSet    psextended {}
psobject      MemberSet    psobject {BaseObject, Members,...
Drive         Method       int Drive(int miles)
Equals        Method       bool Equals(System.Object obj)
GetHashCode   Method       int GetHashCode()
GetType       Method       type GetType()
get_Color     Method       string get_Color()
get_Distance  Method       int get_Distance()
get_ModelYear Method       string get_ModelYear()
get_rides     Method       int get_rides()
set_Color     Method       void set_Color(string )
set_Distance  Method       void set_Distance(int )
```

```
set_ModelYear Method      void set_ModelYear(string )
set_rides     Method      void set_rides(int )
ToString      Method      string ToString()
Color         Property    string Color {get;set;}
Distance      Property    int Distance {get;set;}
ModelYear     Property    string ModelYear {get;set;}
rides         Property    int rides {get;set;}
```

SEE ALSO

about_Classes
about_Language_Keywords
about_Wildcards
Get-Member

TOPIC

about_History

SHORT DESCRIPTION

Describes how to get and run commands in the command history.

LONG DESCRIPTION

When you enter a command at the command prompt, Windows PowerShell saves the command in the command history. You can use the commands in the history as a record of your work. And, you can recall and run the commands from the command history.

History Cmdlets

Windows PowerShell has a set of cmdlets that manage the command history.

Cmdlet (Alias) Description

Get-History (h)	Gets the command history.
Invoke-History (r)	Runs a command in the command history.
Add-History	Adds a command to the command history.
Clear-History (clh)	Deletes commands from the command history.

Keyboard Shortcuts for Managing History

In the Windows PowerShell console, you can use the following shortcuts to manage the command history.

For other host applications, see the product documentation.

Use this key	To perform this action
UP ARROW	Displays the previous command.
DOWN ARROW	Displays the next command.
F7	Displays the command history. To hide the history, press ESC.
F8	Finds a command. Type one or more characters, and then press F8. For the next instance, press F8 again.
F9	Find a command by history ID. Type the history ID, and then press F9. To find the ID, press F7.

MaximumHistoryCount

The $MaximumHistoryCount preference variable determines the maximum number of commands that Windows PowerShell saves in the command history. The default value is 4096, meaning that Windows PowerShell saves the 4096 most recent commands, but you can change the value of the variable.

For example, the following command lowers the $MaximumHistoryCount to 100 commands:

```
$MaximumHistoryCount = 100
```

To apply the setting, restart Windows PowerShell.

To save the new variable value for all your Windows PowerShell sessions, add the assignment statement to a Windows PowerShell profile. For more information about profiles, see about_Profiles (http://go.microsoft.com/fwlink/?LinkID=113729).

For more information about the $MaximumHistoryCount preference variable, see about_Preference_Variables (http://go.microsoft.com/fwlink/?LinkID=113248).

NOTE: In Windows PowerShell 2.0, the default value of the $MaximumHistoryCount preference variable is 64.

Order of Commands in the History

Commands are added to the history when the command finishes executing, not when the command is entered. If commands take some time to be completed, or if the commands are executing in a nested prompt, the commands might appear to be out of order in the history. (Commands that are executing in a nested prompt are completed only when you exit the prompt level.)

SEE ALSO

about_Line_Editing
about_Preference_Variables
about_Profiles
about_Variables

TOPIC

about_If

SHORT DESCRIPTION

Describes a language command you can use to run statement lists based on the results of one or more conditional tests.

LONG DESCRIPTION

You can use the If statement to run code blocks if a specified conditional test evaluates to true. You can also specify one or more additional conditional tests to run if all the prior tests evaluate to false. Finally, you can specify an additional code block that is run if no other prior conditional test evaluates to true.

Syntax

The following example shows the If statement syntax:

```
if (<test1>)
   {<statement list 1>}
[elseif (<test2>)
   {<statement list 2>}]
[else
   {<statement list 3>}]
```

When you run an If statement, Windows PowerShell evaluates the <test1> conditional expression as true or false. If <test1> is true, <statement list 1> runs, and Windows PowerShell exits the If statement. If <test1> is false, Windows PowerShell evaluates the condition specified by the <test2> conditional statement.

If <test2> is true, <statement list 2> runs, and Windows PowerShell exits the If statement. If both <test1> and <test2> evaluate to false, the <statement list 3> code block runs, and Windows PowerShell exits the If statement.

You can use multiple Elseif statements to chain a series of conditional tests so that each test is run only if all the previous tests are false. If you need to create an If statement that contains many Elseif statements, consider using a Switch statement instead.

Examples

The simplest If statement contains a single command and does not contain any Elseif statements or any Else statements. The following example shows the simplest form of the If statement:

```
if ($a -gt 2)
{
```

```
    Write-Host "The value $a is greater than 2."
  }
```

In this example, if the $a variable is greater than 2, the condition evaluates to true, and the statement list runs. However, if $a is less than or equal to 2 or is not an existing variable, the If statement does not display a message. By adding an Else statement, a message is displayed when $a is less than or equal to 2, as the next example shows:

```
  if ($a -gt 2)
  {
    Write-Host "The value $a is greater than 2."
  }
  else
  {
    Write-Host "The value $a is less than or equal to 2, is not
  created or is not initialized."
  }
```

To further refine this example, you can use the Elseif statement to display a message when the value of $a is equal to 2, as the next example shows:

```
  if ($a -gt 2)
  {
    Write-Host "The value $a is greater than 2."
  }
  elseif ($a -eq 2)
  {
    Write-Host "The value $a is equal to 2."
  }
  else
  {
    Write-Host "The value $a is less than 2 or was not created
  or initialized."
  }
```

SEE ALSO

about_Comparison_Operators
about_Switch

Name	Category	Module	Synopsis
about_InlineScript	HelpFile		Describes the InlineScript activity, which runs Windows
about_InlineScript	HelpFile		Describes the InlineScript activity, which runs Windows

TOPIC

about_Jobs

SHORT DESCRIPTION

Provides information about how Windows PowerShell background jobs run a command or expression in the background without interacting with the current session.

LONG DESCRIPTION

This topic explains how to run background jobs in Windows PowerShell on a local computer. For information about running background jobs on remote computers, see about_Remote_Jobs.

When you start a background job, the command prompt returns immediately, even if the job takes an extended time to complete. You can continue to work in the session without interruption while the job runs.

THE JOB CMDLETS

Start-Job Starts a background job on a local computer.

Get-Job Gets the background jobs that were started in the current session.

Receive-Job Gets the results of background jobs.

Stop-Job Stops a background job.

Wait-Job Suppresses the command prompt until one or all jobs are complete.

Remove-Job Deletes a background job.

Invoke-Command The AsJob parameter runs any command as a background job on a remote computer. You can also use Invoke-Command to run any job command remotely, including a Start-Job command.

HOW TO START A JOB ON THE LOCAL COMPUTER

To start a background job on the local computer, use the Start-Job cmdlet.

To write a Start-Job command, enclose the command that the job runs in braces ({ }). Use the ScriptBlock parameter to specify the command.

The following command starts a background job that runs a Get-Process command on the local computer.

```
Start-Job -ScriptBlock {Get-Process}
```

The Start-Job command returns an object that represents the job. The job object contains useful information about the job, but it does not contain the job results.

Save the job object in a variable, and then use it with the other Job cmdlets to manage the background job. The following command starts a job object and saves the resulting job object in the $job variable.

```
$job = Start-Job -ScriptBlock {Get-Process}
```

You can also use the Get-Job cmdlet to get objects that represent the jobs started in the current session. Get-Job returns the same job object that Start-Job returns.

GETTING JOB OBJECTS

To get object that represent the background jobs that were started in the current session, use the Get-Job cmdlet. Without parameters, Get-Job

returns all of the jobs that were started in the current session.

For example, the following command gets the jobs in the current session.

```
PS C:\>Get-Job

Id  Name  PSJobTypeName State     HasMoreData  Location   Command
--  ----  ------------- -----     -----------  --------   -------
1   Job1  BackgroundJob Running   True         localhost  Get-Process
```

You can also save the job object in a variable and use it to represent the job in a later command. The following command gets the job with ID 1 and saves it in the $job variable.

```
$job = Get-Job -Id 1
```

The job object contains the state of the job, which indicates whether the job has finished. A finished job has a state of "Complete" or "Failed". A job might also be blocked or running.

```
Get-Job

Id  Name  PSJobTypeName State     HasMoreData  Location   Command
--  ----  ------------- -----     -----------  --------   -------
1   Job1  BackgroundJob Complete  True         localhost  Get-Process
```

GETTING THE RESULTS OF A JOB

When you run a background job, the results do not appear immediately. Instead, the Start-Job cmdlet returns a job object that represents the job, but it does not contain the results. To get the results of a background job, use the Receive-Job cmdlet.

The following command uses the Receive-Job cmdlet to get the results of the job. It uses a job object saved in the $job variable to identify the job.

```
Receive-Job -Job $job
```

The Receive-Job cmdlet returns the results of the job.

```
      Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)    Id ProcessName
      -------  ------    -----      ----- -----   ------    -- -----------
          103       4    11328       9692    56          1176 audiodg
          804      14    12228      14108   100   101.74  1740 CcmExec
          668       7     2672       6168   104    32.26   488 csrss
...
```

You can also save the results of a job in a variable. The following command saves the results of the job in the $job variable to the $results variable.

```
$results = Receive-Job -Job $job
```

And, you can save the results of the job in a file by using the redirection operator (>) or the Out-File cmdlet. The following command uses the redirection operator to save the results of the job in the $job variable in the Results.txt file.

```
Receive-Job -Job $job > results.txt
```

GETTING AND KEEPING PARTIAL JOB RESULTS

The Receive-Job cmdlet gets the results of a background job. If the job is complete, Receive-Job gets all job results. If the job is still running, Receive-Job gets the results that have been generated thus far. You can run Receive-Job commands again to get the remaining results.

When Receive-Job returns results, by default, it deletes those results from the cache where job results are stored. If you run another Receive-Job command, you get only the results that are not yet received.

The following commands show the results of Receive-Job commands run before the job is complete.

```
C:\PS> Receive-Job -Job $job

      Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
      -------  ------    -----      ----- -----   ------     -- -----------
          103       4    11328       9692    56          1176 audiodg
          804      14    12228      14108   100   101.74   1740 CcmExec


C:\PS> Receive-Job -Job $job
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     68       3     2632       664    29     0.36   1388 ccmsetup
    749      22    21468     19940   203   122.13   3644 communicator
    905       7     2980      2628    34   197.97    424 csrss
   1121      25    28408     32940   174   430.14   3048 explorer
```

To prevent Receive-Job from deleting the job results that it has returned, use the Keep parameter. As a result, Receive-Job returns all of the results that have been generated until that time.

The following commands show the effect of using the Keep parameter on a job that is not yet complete.

```
C:\PS> Receive-Job -Job $job -Keep

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    103       4    11328      9692    56          1176 audiodg
    804      14    12228     14108   100   101.74   1740 CcmExec
```

```
C:\PS> Receive-Job -Job $job -Keep

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    103       4    11328      9692    56          1176 audiodg
    804      14    12228     14108   100   101.74   1740 CcmExec
     68       3     2632       664    29     0.36   1388 ccmsetup
    749      22    21468     19940   203   122.13   3644 communicator
    905       7     2980      2628    34   197.97    424 csrss
   1121      25    28408     32940   174   430.14   3048 explorer
```

WAITING FOR THE RESULTS

If you run a command that takes a long time to complete, you can use the properties of the job object to determine when the job is complete. The following command uses the Get-Job object to get all of the background jobs in the current session.

```
Get-Job
```

The results appear in a table. The status of the job appears in the State column.

```
Id  Name  PSJobTypeName State      HasMoreData  Location   Command
--  ----  ------------- -----      -----------  --------   -------
1   Job1  BackgroundJob Complete    True         localhost  Get-Process
2   Job2  BackgroundJob Running     True         localhost  Get-EventLog -Log Syst...
3   Job3  BackgroundJob Complete    True         localhost  dir -Path C:\* -Recurse
```

In this case, the State property reveals that Job 2 is still running. If you were to use the Receive-Job cmdlet to get the job results now, the results would be incomplete. You can use the Receive-Job cmdlet repeatedly to get all of the results. By default, each time you use it, you get only the results that were not already received, but you can use the Keep parameter of the Receive-Job cmdlet to retain the results, even though they were already received.

You can write the partial results to a file and then append newer results as they arrive or you can wait and check the state of the job later.

You can use the Wait parameter of the Receive-Job cmdlet, which does not return the command prompt until the job is complete and all results are available.

You can also use the Wait-Job cmdlet to wait for any or all of the results of the job. Wait-Job lets you wait for a particular job, for all jobs, or for any of the jobs to be completed.

The following command uses the Wait-Job cmdlet to wait for a job with ID 10.

```
Wait-Job -ID 10
```

As a result, the Windows PowerShell prompt is suppressed until the job is completed.

You can also wait for a predetermined period of time. This command uses the Timeout parameter to limit the wait to 120 seconds. When the time expires, the command prompt returns, but the job continues to run in the background.

```
Wait-Job -ID 10 -Timeout 120
```

STOPPING A JOB

To stop a background job, use the Stop-Job cmdlet. The following command starts a job to get every entry in the System event log. It saves the job object in the $job variable.

```
$job = Start-Job -ScriptBlock {Get-EventLog -Log System}
```

The following command stops the job. It uses a pipeline operator (|) to send the job in the $job variable to Stop-Job.

```
$job | Stop-Job
```

DELETING A JOB

To delete a background job, use the Remove-Job cmdlet. The following command deletes the job in the $job variable.

```
Remove-Job -Job $job
```

INVESTIGATING A FAILED JOB

To find out why a job failed, use the Reason subproperty of the job object.

The following command starts a job without the required credentials. It saves the job object in the $job variable.

```
$job = Start-Job -ScriptBlock {New-Item -Path HKLM:\Software\MyCompany}

Id   Name  PSJobTypeName State    HasMoreData  Location   Command
--   ----  ------------- -----    -----------  --------   -------
1    Job1  BackgroundJob Failed   False        localhost  New-Item -Path HKLM:\S...
```

The following command uses the Reason property to find the error that caused the job to fail.

```
$job.ChildJobs[0].JobStateInfo.Reason
```

In this case, the job failed because the remote computer required explicit credentials to run the command. The value of the Reason property is:

```
Connecting to remote server failed with the following error
message : Access is denied.
```

SEE ALSO
about_Remote_Jobs
about_Job_Details
about_Remote
about_PSSessions
Start-Job
Get-Job
Receive-Job
Stop-Job
Wait-Job
Remove-Job
Invoke-Command

TOPIC

about_Job_Details

SHORT DESCRIPTION
Provides details about background jobs on local and remote computers.

DETAILED DESCRIPTION
This topic explains the concept of a background job and provides technical information about how background jobs work in Windows PowerShell.

This topic is a supplement to the about_Jobs and about_Remote_Jobs topics.

ABOUT BACKGROUND JOBS
A background job runs a command or expression asynchronously. It might run a cmdlet, a function, a script, or any other command-based task. It is designed to run commands that take an extended period of time, but you can use it to run any command in the background.

When a synchronous command runs, the Windows PowerShell command prompt is suppressed until the command is complete. But a background job does not suppress the Windows PowerShell prompt. A command to start a background job

returns a job object. The prompt returns immediately so you can work on other tasks while the background job runs.

However, when you start a background job, you do not get the results immediately even if the job runs very quickly. The job object that is returned contains useful information about the job, but it does not contain the job results. You must run a separate command to get the job results. You can also run commands to stop the job, to wait for the job to be completed, and to delete the job.

To make the timing of a background job independent of other commands, each background job runs in its own Windows PowerShell environment (a "session"). However, this can be a temporary connection that is created only to run the job and is then destroyed, or it can be a persistent session (a PSSession) that you can use to run several related jobs or commands.

USING THE JOB CMDLETS

Use a Start-Job command to start a background job on a local computer. Start-Job returns a job object. You can also get objects representing the jobs that were started on the local computer by using the Get-Job cmdlet.

To get the job results, use a Receive-Job command. If the job is not complete, Receive-Job returns partial results. You can also use the Wait-Job cmdlet to suppress the command prompt until one or all of the jobs that were started in the session are complete.

To stop a background job, use the Stop-Job cmdlet. To delete a job, use the Remove-Job cmdlet.

For more information about how the cmdlets work, see the Help topic for each cmdlet, and see about_Jobs.

STARTING BACKGROUND JOBS ON REMOTE COMPUTERS

You can create and manage background jobs on a local or remote computer. To run a background job remotely, use the AsJob parameter of a cmdlet such as Invoke-Command, or use the Invoke-Command cmdlet to run a Start-Job command remotely. You can also start a background job in an interactive session.

For more information about remote background jobs, see about_Remote_Jobs.

CHILD JOBS

Each background job consists of a parent job and one or more child jobs. In jobs started by using Start-Job or the AsJob parameter of Invoke-Command,

the parent job is an executive. It does not run any commands or return any results. The commands are actually run by the child jobs. (Jobs started by using other cmdlets might work differently.)

The child jobs are stored in the ChildJobs property of the parent job object. The ChildJobs property can contain one or many child job objects. The child job objects have a name, ID, and instance ID that differ from the parent job so that you can manage the parent and child jobs individually or as a unit.

To get the parent and child jobs of a job, use the IncludeChildJobs parameter of the Get-Job cmdlet. The IncludeChildJob parameter is introduced in Windows PowerShell 3.0.

```
C:\PS> Get-Job -IncludeChildJob

Id   Name   PSJobTypeName State     HasMoreData   Location    Command
--   ----   ------------- -----     -----------   --------    -------
1    Job1   RemoteJob     Failed    True          localhost   Get-Process
2    Job2                 Completed True          Server01    Get-Process
3    Job3                 Failed    False         localhost   Get-Process
```

To get the parent job and only the child jobs with a particular State value, use the ChildJobState parameter of the Get-Job cmdlet. The ChildJobState parameter is introduced in Windows PowerShell 3.0.

```
C:\PS> Get-Job -ChildJobState Failed

Id   Name   PSJobTypeName State     HasMoreData   Location    Command
--   ----   ------------- -----     -----------   --------    -------
1    Job1   RemoteJob     Failed    True          localhost   Get-Process
3    Job3                 Failed    False         localhost   Get-Process
```

To get the child jobs of a job on all versions of Windows PowerShell, use the ChildJob property of the parent job.

```
C:\PS> (Get-Job Job1).ChildJobs

Id   Name   PSJobTypeName State     HasMoreData   Location    Command
--   ----   ------------- -----     -----------   --------    -------
2    Job2                 Completed True          Server01    Get-Process
3    Job3                 Failed    False         localhost   Get-Process
```

You can also use a Get-Job command on the child job, as shown in the

following command:

```
C:\PS> Get-Job Job3

Id    Name   PSJobTypeName State     HasMoreData   Location    Command
--    ----   ------------- -----     -----------   --------    -------
3     Job3                 Failed    False         localhost   Get-Process
```

The configuration of the child job depends on the command that you use to start the job.

-- When you use Start-Job to start a job on a local computer, the job consists of an executive parent job and a child job that runs the command.

-- When you use the AsJob parameter of Invoke-Command to start a job on one or more computers, the job consists of an executive parent job and a child job for each job run on each computer.

-- When you use Invoke-Command to run a Start-Job command on one or more remote computers, the result is the same as a local command run on each remote computer. The command returns a job object for each computer. The job object consists of an executive parent job and one child job that runs the command.

The parent job represents all of the child jobs. When you manage a parent job, you also manage the associated child jobs. For example, if you stop a parent job, all child jobs are stopped. If you get the results of a parent job, you get the results of all child jobs.

However, you can also manage child jobs individually. This is most useful when you want to investigate a problem with a job or get the results of only one of a number of child jobs started by using the AsJob parameter of Invoke-Command. (The backtick character [`] is the continuation character.)

The following command uses the AsJob parameter of Invoke-Command to start background jobs on the local computer and two remote computers. The command saves the job in the $j variable.

```
PS C:> $j = Invoke-Command -ComputerName localhost, Server01, Server02 `
     -Command {Get-Date} -AsJob
```

When you display the Name and ChildJob properties of the job in $j, it shows that the command returned a job object with three child jobs, one for each computer.

```
CPS C:> $j | Format-List Name, ChildJobs

Name      : Job3
ChildJobs : {Job4, Job5, Job6}
```

When you display the parent job, it shows that the job failed.

```
C:\PS> $j

Id    Name   PSJobTypeName State     HasMoreData   Location
--    ----   ------------- -----     -----------   --------
3     Job3   RemotingJob   Failed    False         localhost,Server...
```

But when you run a Get-Job command that gets the child jobs, the output shows that only one child job failed.

```
PS C:\> Get-Job -IncludeChildJobs

Id    Name   PSJobTypeName State     HasMoreData   Location    Command
--    ----   ------------- -----     -----------   --------    -------
3     Job3   RemotingJob   Failed    False         localhost,Server...
4     Job4                 Completed True          localhost   Get-Date
5     Job5                 Failed    False         Server01    Get-Date
6     Job6                 Completed True          Server02    Get-Date
```

To get the results of all child jobs, use the Receive-Job cmdlet to get the results of the parent job. But you can also get the results of a particular child job, as shown in the following command.

```
C:\PS> Receive-Job -Name Job6 -Keep | Format-Table ComputerName, DateTime -Auto

ComputerName DateTime
------------ --------
Server02     Thursday, March 13, 2008 4:16:03 PM
```

The child jobs feature of Windows PowerShell background jobs gives you more control over the jobs that you run.

JOB TYPES

Windows PowerShell supports different types of jobs for different tasks. Beginning in Windows PowerShell 3.0, developers can write "job source adapters" that add new job types to Windows PowerShell and include the

job source adapters in modules. When you import the module, you can use the new job type in your session.

For example, the PSScheduledJob module adds scheduled jobs and the PSWorkflow module adds workflow jobs.

Custom jobs types might differ significantly from standard Windows PowerShell background jobs. For example, scheduled jobs are saved on disk; they do not exist only in a particular session. Workflow jobs can be suspended and resumed.

The cmdlets that you use to manage custom jobs depend on the job type. For some, you use the standard job cmdlets, such as Get-Job and Start-Job. Others come with specialized cmdlets that manage only a particular type of job. For detailed information about custom job types, see the help topics about the job type.

To find the job type of a job, use the Get-Job cmdlet. Get-Job returns different job objects for different types of jobs. The value of the PSJobTypeName property of the job objects that Get-Job returns indicates the job type.

The following table lists the job types that come with Windows PowerShell.

Job Type	Description
BackgroundJob	Started by using the Start-Job cmdlet.
RemoteJob	Started by using the AsJob parameter of the Invoke-Command cmdlet.
PSWorkflowJob	Started by using the AsJob parameter of a workflow.
PSScheduledJob	An instance of a scheduled job started by a job trigger.
CIMJob	Started by using the AsJob parameter of a cmdlet from a CDXML module.
WMIJob	Started by using the AsJob parameter of a cmdlet from a WMI module.
PSEventJob	Created by running Register-ObjectEvent and specifying an action with the Action parameter.

NOTE: Before using the Get-Job cmdlet to get jobs of a particular type, verify that the module that adds the job type is imported into the current session. Otherwise, Get-Job does not get jobs of that type.

EXAMPLE

```
  The following commands create a local background job, a
  remote background job, a workflow job, and a scheduled job.
  Then, it uses the Get-Job cmdlet to get the jobs. Get-Job
  does not get the scheduled job, but it gets any started
  instances of the scheduled job.

  # Start a background job on the local computer.
  PS C:\> Start-Job -Name LocalData {Get-Process}

  Id   Name      PSJobTypeName   State   HasMoreData    Location   Command
  --   ----      -------------   -----   -----------    --------   -------
  2    LocalData   BackgroundJob   Running      True    localhost  Get-Process



  # Start a background job that runs on a remote computer.
  PS C:\> Invoke-Command -ComputerName Server01 {Get-Process} -AsJob -JobName RemoteData

  Id   Name      PSJobTypeName   State   HasMoreData    Location   Command
  --   ----      -------------   -----   -----------    --------   -------
  2    RemoteData  RemoteJob      Running      True    Server01   Get-Process



  # Create a scheduled job
  PS C:\>  Register-ScheduledJob -Name ScheduledJob -ScriptBlock {Get-Process} `
         -Trigger (New-JobTrigger -Once -At "3 PM")

  Id       Name          JobTriggers     Command      Enabled
  --       ----          -----------     -------      -------
  1        ScheduledJob    1             Get-Process   True



  # Create a workflow.
  PS C:\> workflow Test-Workflow {Get-Process}

  # Run the workflow as a job.
  PS C:\> Test-Workflow -AsJob -JobName TestWFJob

  Id   Name      PSJobTypeName   State   HasMoreData    Location   Command
  --   ----      -------------   -----   -----------    --------   -------
  2    TestWFJob   PSWorkflowJob   NotStarted    True    localhost  Get-Process
```

```
# Get the jobs. The Get-Job command does not get scheduled jobs, but it gets
  instances of the scheduled job that are started.

PS C:\> Get-Job

Id   Name       PSJobTypeName   State     HasMoreData     Location  Command
--   ----       -------------   -----     -----------     --------  -------
2    LocalData    BackgroundJob   Completed True          localhost Get-Process
4    RemoteData   RemoteJob       Completed True          Server01  Get-Process
6    TestWFJob    PSWorkflowJob   Completed True          localhost WorkflowJob
8    ScheduledJob PSScheduledJob  Completed True          localhost Get-Process

# To get scheduled jobs, use the Get-ScheduledJob cmdlet.
PS C:\> Get-ScheduledJob

Id       Name          JobTriggers     Command      Enabled
--       ----          -----------     -------      -------
1        ScheduledJob    1             Get-Process   True
```

SEE ALSO

about_Jobs
about_Remote_Jobs
about_Remote
about_Scheduled_Jobs
Invoke-Command
Start-Job
Get-Job
Wait-Job
Stop-Job
Remove-Job
Suspend-Job
Resume-Job
New-PSSession
Enter-PSSession
Exit-PSSession
Register-ScheduledJob
Get-ScheduleJob

TOPIC

about_join

SHORT DESCRIPTION

Describes how the join operator (-join) combines multiple strings into a single string.

LONG DESCRIPTION

The join operator concatenates a set of strings into a single string. The strings are appended to the resulting string in the order that they appear in the command.

Syntax

The following diagram shows the syntax for the join operator.

```
-Join <String[]>
<String[]> -Join <Delimiter>
```

Parameters

String[]

Specifies one or more strings to be joined.

Delimiter

Specifies one or more characters placed between the concatenated strings. The default is no delimiter ("").

Remarks

The unary join operator (-join <string[]>) has higher precedence than a comma. As a result, if you submit a comma-separated list of strings to the unary join operator, only the first string (before the first comma) is submitted to the join operator.

To use the unary join operator, enclose the strings in parentheses, or store the strings in a variable, and then submit the variable to join.

For example:

```
-join "a", "b", "c"
a
b
c

-join ("a", "b", "c")
abc


$z = "a", "b", "c"
-join $z
abc
```

Examples

The following statement joins three strings:

```
-join ("Windows", "PowerShell", "2.0")
WindowsPowerShell2.0
```

The following statement joins three strings delimited by a space:

```
"Windows", "PowerShell", "2.0" -join " "
Windows PowerShell 2.0
```

The following statements use a multiple-character delimiter to join three strings:

```
$a = "WIND", "SP", "ERSHELL"
$a -join "OW"
WINDOWSPOWERSHELL
```

The following statement joins the lines in a here-string into a single string. Because a here-string is one string, the lines in the here-string must be split before they can be joined. You can use this method to rejoin the strings in an XML file that has been saved in a here-string:

```
$a = @'
a
```

```
b
c
'@

(-split $a) -join " "
a b c
```

SEE ALSO

about_Operators
about_Comparison_Operators
about_Split

TOPIC

about_Language_Keywords

SHORT DESCRIPTION

Describes the keywords in the Windows PowerShell scripting language.

LONG DESCRIPTION

Windows PowerShell has the following language keywords. For more information, see the about topic for the keyword and the information that follows the table.

Keyword	Reference
Begin	about_Functions, about_Functions_Advanced
Break	about_Break, about_Trap
Catch	about_Try_Catch_Finally
Continue	about_Continue, about_Trap
Data	about_Data_Sections
Do	about_Do, about_While
DynamicParam	about_Functions_Advanced_Parameters

Else	about_If
Elseif	about_If
End	about_Functions, about_Functions_Advanced_Methods
Exit	Described in this topic.
Filter	about_Functions
Finally	about_Try_Catch_Finally
For	about_For
ForEach	about_ForEach
From	Reserved for future use.
Function	about_Functions, about_Functions_Advanced
If	about_If
In	about_ForEach
InlineScript	about_InlineScript
Hidden	about_Hidden
Parallel	about_Parallel, about_ForEach-Parallel
Param	about_Functions
Process	about_Functions, about_Functions_Advanced
Return	about_Return
Sequence	about_Sequence
Switch	about_Switch
Throw	about_Throw, about_Functions_Advanced_Methods
Trap	about_Trap, about_Break, about_Try_Catch_Finally
Try	about_Try_Catch_Finally
Until	about_Do
While	about_While, about_Do
Workflow	about_Workflows

Language Keywords

Begin

Specifies one part of the body of a function, along with the DynamicParam, Process, and End keywords. The Begin statement list runs one time before any objects are received from the pipeline.

Syntax:

```
function <name> {
   DynamicParam {<statement list>}
   begin {<statement list>}
   process {<statement list>}
   end {<statement list>}
}
```

Break

Causes a script to exit a loop.

Syntax:

```
while (<condition>) {
   <statements>
      ...
   break
      ...
   <statements>
}
```

Catch

Specifies a statement list to run if an error occurs in the accompanying Try statement list. An error type requires brackets. The second pair of brackets indicates that the error type is optional.

Syntax:

```
try {<statement list>}
catch [[<error type>]] {<statement list>}
```

Continue

Causes a script to stop running a loop and to go back to the condition. If the condition is met, the script begins the loop again.

Syntax:

```
while (<condition>) {
   <statements>
      ...
   continue
      ...
   <statements>
}
```

Data

In a script, defines a section that isolates data from the script logic. Can also include If statements and some limited commands.

Syntax:

data <variable> [-supportedCommand <cmdlet-name>] {<permitted content>}

Do
--

Used with the While or Until keyword as a looping construct. Windows PowerShell runs the statement list at least one time, unlike a loop that uses While.

Syntax:

```
do {<statement list>} while (<condition>)

do {<statement list>} until (<condition>)
```

DynamicParam

Specifies one part of the body of a function, along with the Begin, Process, and End keywords. Dynamic parameters are added at run time.

Syntax:

```
function <name> {
   DynamicParam {<statement list>}
   begin {<statement list>}
   process {<statement list>}
   end {<statement list>}
}
```

Else

Used with the If keyword to specify the default statement list.

Syntax:

```
if (<condition>) {<statement list>}
else {<statement list>}
```

Elseif

Used with the If and Else keywords to specify additional conditionals. The Else keyword is optional.

Syntax:

```
   if (<condition>) {<statement list>}
   elseif (<condition>) {<statement list>}
   else {<statement list>}

End
```

Specifies one part of the body of a function, along with the DynamicParam, Begin, and End keywords. The End statement list runs one time after all the objects have been received from the pipeline.

Syntax:

```
   function <name> {
      DynamicParam {<statement list>}
      begin {<statement list>}
      process {<statement list>}
      end {<statement list>}
   }

Exit
```

Causes Windows PowerShell to exit a script or a Windows PowerShell instance.

When you run 'powershell.exe –File <path to a script>', you can only set the %ERRORLEVEL% variable to a value other than zero by using the exit statement. In the following example, the user sets the error level variable value to 4 by typing 'exit 4'.

```
   C:\Users\bruce\documents\test>type test.ps1
   1
   2
   3
   exit 4

   C:\Users\bruce\documents\test>powershell -file ./test.ps1
   1
   2
   3

   C:\Users\bruce\documents\test>echo %ERRORLEVEL%
   4
```

When you use powershell.exe with the File parameter, the .ps1 (script) file itself should include instructions for handling any errors or

exceptions that occur while the script is running. You should only use the exit statement to indicate the post-execution status of the script.

Syntax:

```
exit
exit <exit code>
```

Filter

Specifies a function in which the statement list runs one time for each input object. It has the same effect as a function that contains only a Process block.

Syntax:

```
filter <name> {<statement list>}
```

Finally

Defines a statement list that runs after statements that are associated with Try and Catch. A Finally statement list runs even if you press CTRL+C to leave a script or if you use the Exit keyword in the script.

Syntax:

```
try {<statement list>}
catch [<error type] {<statement list>}
finally {<statement list>}
```

For

Defines a loop by using a condition.

Syntax:

```
for (<initialize>; <condition>; <iterate>) {<statement list>}
```

ForEach

Defines a loop by using each member of a collection.

Syntax:

```
ForEach (<item> in <collection>){<statement list>}
```

From

Reserved for future use.

Function

Creates a named statement list of reusable code. You can name the scope a function belongs to. And, you can specify one or more named parameters by using the Param keyword. Within the function statement list, you can include DynamicParam, Begin, Process, and End statement lists.

Syntax:

```
function [<scope:>]<name> {
    param ([type]<$pname1> [, [type]<$pname2>])
    DynamicParam {<statement list>}
    begin {<statement list>}
    process {<statement list>}
    end {<statement list>}
}
```

You also have the option of defining one or more parameters outside the statement list after the function name.

Syntax:

```
function [<scope:>]<name> [([type]<$pname1>, [[type]<$pname2>])] {
    DynamicParam {<statement list>}
    begin {<statement list>}
    process {<statement list>}
    end {<statement list>}
}
```

If
--

Defines a conditional.

Syntax:

```
if (<condition>) {<statement list>}
```

Hidden

Hides class members from the default results of the Get-Member cmdlet, and from IntelliSense and tab completion results.

Syntax:

Hidden [data type] $member_name

In
--

Used in a ForEach statement to create a loop that uses each member of a collection.

Syntax:

```
ForEach (<item> in <collection>){<statement list>}
```

InlineScript

Runs workflow commands in a shared Windows PowerShell session.
This keyword is valid only in a Windows PowerShell Workflow.

Syntax:

```
workflow <verb>-<noun>
{
  InlineScript
  {
    <Command/Expression>
    ...
  }
}
```

The InlineScript keyword indicates an InlineScript activity, which runs commands in a shared standard (non-workflow) session. You can use the InlineScript keyword to run commands that are not otherwise valid in a workflow, and to run commands that share data. By default, the commands in an InlineScript script block run in a separate process.

For more information, see about_InlineScript and Running Windows PowerShell Commands in a Workflow (http://technet.microsoft.com/library/jj574197.aspx).

Param

Defines the parameters in a function.

Syntax:

```
function [<scope:>]<name> {
   param ([type]<$pname1>[, [[type]<$pname2>]])
   <statement list>
}
```

Parallel

Runs workflow commands concurrently and in an undefined order.
This keyword is valid only in a Windows PowerShell Workflow.

The Parallel keyword indicates a Parallel script block. The commands in a Parallel script block can run at the same time and in any order. This feature significantly improves the performance of a workflow.

Syntax:

```
workflow <verb>-<noun>
{
   Parallel
   {
      <Activity>
      <Activity>
      …
   }
}
```

The Parallel parameter of the ForEach keyword processes the items in a collection in parallel. The activities in the script block run sequentially on each item, but the script block can run on multiple items at the same time and the items are processed in an undefined order.

Syntax:

```
workflow <verb>-<noun>
{
   ForEach -Parallel (<item> in <collection>)
   {
      <Activity>
```

```
        <Activity>
        ...
    }
}
```

For more information, see about_Parallel, about_ForEach-Parallel

Process

Specifies a part of the body of a function, along with the DynamicParam, Begin, and End keywords. When a Process statement list receives input from the pipeline, the Process statement list runs one time for each element from the pipeline. If the pipeline provides no objects, the Process statement list does not run. If the command is the first command in the pipeline, the Process statement list runs one time.

Syntax:

```
function <name> {
    DynamicParam {<statement list>}
    begin {<statement list>}
    process {<statement list>}
    end {<statement list>}
}
```

Return

Causes Windows PowerShell to leave the current scope, such as a script or function, and writes the optional expression to the output.

Syntax:

```
return [<expression>]
```

Sequence

Runs workflow commands sequentially in a Parallel script block.
This keyword is valid only in a Windows PowerShell Workflow.

Syntax:

```
workflow <verb>-<noun>
```

```
{
   Parallel
   {
      Sequence
      {
         <Activity>
      }
   }
}
```

The Sequence keyword creates a sequence block within a Parallel script block. The commands in the Sequence script block run sequentially and in the order defined.

For more information, see about_Sequence

Switch

Specifies a variety of actions to be performed on items from the pipeline or from a file. You can use either of the following syntax models.

Syntax 1:

```
   switch [-regex|-wildcard|-exact][-casesensitive] ( pipeline )

   {
      <string>|<number>|<variable>|{ <expression> } {<statement list>}
      <string>|<number>|<variable>|{ <expression> } {<statement list>}
           ...
      default {<statement list>}
   }
```

Syntax 2:

```
   switch [-regex|-wildcard|-exact][-casesensitive] -file filename
   {
      <string>|<number>|<variable>|{ <expression> } {<statement list>}
      <string>|<number>|<variable>|{ <expression> } {<statement list>}
           ...
      default {<statement list>}
   }
```

Throw

Throws an object as an error.

Syntax:

```
throw [<object>]
```

Trap

Defines a statement list to be run if an error is encountered. An error type requires brackets. The second pair of brackets indicates that the error type is optional.

Syntax:

```
trap [[<error type>]] {<statement list>}
```

Try

Defines a statement list to be checked for errors while the statements run. If an error occurs, Windows PowerShell continues running in a Catch or Finally statement. An error type requires brackets. The second pair of brackets indicates that the error type is optional.

Syntax:

```
try {<statement list>}
catch [[<error type]] {<statement list>}
finally {<statement list>}
```

Until

Used in a Do statement as a looping construct where the statement list is executed at least one time.

Syntax:

```
do {<statement list>} until (<condition>)
```

While

Used in a Do statement as a looping construct where the statement list is executed at least one time.

Syntax:

```
do {<statement list>} while (<condition>)
```

Workflow

Creates a script-based Windows PowerShell workflow, that is, a workflow written in the Windows PowerShell language.

A Windows PowerShell workflow is a Windows PowerShell command type that is supported by Windows PowerShell and Windows Workflow Foundation. Workflows are designed for complex, long-running tasks that affect multiple computers. Workflows can be recovered if interrupted, such as by a network outage, and you can suspend and resume them without losing state or data.

Workflows can be written in XAML, the native language of Windows Workflow Foundation, or in the Windows PowerShell language.

The syntax of a script-based workflow is similar to the syntax of a function. However, the unit of execution in a workflow is an activity, instead of a command. Cmdlets (and other commands) that are used in script-based workflows are implicitly converted to activities.

Some language elements that are permitted in scripts and functions are not permitted in workflows. Similarly, workflows can include elements that are not found in scripts and functions, such as "persistence points" (checkpoints), self-suspension, and parallel processing. In addition, all workflows have a set of common parameters that are added by Windows PowerShell when you use the Workflow keyword.

Syntax:

```
  workflow <Verb-Noun> {
     param ([type]<$pname1> [, [type]<$pname2>])
     <statement list>
  }

  workflow <verb-noun>
  {
     [CmdletBinding(<Attributes>)]
      Param
      (
         [Parameter(<Arguments>)]
         $Param1
      )
```

<statement list>
}

For more information about workflows, see about_Workflows
and "Getting Started with Windows PowerShell Workflow"
(http://go.microsoft.com/fwlink/?LinkID=252592) in the
TechNet Library.

SEE ALSO
about_Escape_Characters
about_Special_Characters
about_Wildcards

TOPIC

about_Language_Modes

SHORT DESCRIPTION
Explains language modes and their effect on Windows
PowerShell sessions.

LONG DESCRIPTION
The language mode of a Windows PowerShell session determines,
in part, which elements of the Windows PowerShell language
can be used in the session.

Windows PowerShell supports the following language modes:
-- FullLanguage
-- ConstrainedLanguage (introduced in Windows PowerShell 3.0)
-- RestrictedLanguage
-- NoLanguage

WHAT IS A LANGUAGE MODE?

The language mode determines the language elements that are permitted in the session.

The language mode is actually a property of the session configuration (or "endpoint") that is used to create the session. All sessions that use a particular session configuration have the language mode of the session configuration.

All Windows PowerShell sessions have a language mode, including PSSessions that you create by using the New-PSSession cmdlet, temporary sessions that use the ComputerName parameter, and the default sessions that appear when you start Windows PowerShell.

Remote sessions are created by using the session configurations on the remote computer. The language mode set in the session configuration determines the language mode of the session. To specify the session configuration of a PSSession, use the ConfigurationName parameter of cmdlets that create a session.

LANGUAGE MODES

This section describes the language modes in Windows PowerShell sessions.

FULL LANGUAGE (FullLanguage)

The FullLanguage language mode permits all language elements in the session. FullLanguage is the default language mode for default sessions on all versions of Windows except for Windows RT.

RESTRICTED LANGUAGE (RestrictedLanguage)

In RestrictedLanguage language mode, users may run commands (cmdlets, functions, CIM commands, and workflows) but are not permitted to use script blocks.

Only the following variables are permitted:
-- $PSCulture
-- $PSUICulture
-- $True
-- $False
-- $Null.

Only the following comparison operators are permitted:
-- -eq (equal)
-- -gt (greater-than)
-- -lt (less-than)

Assignment statements, property references, and method calls are not permitted.

NO LANGUAGE (NoLanguage)

In NoLanguage language mode, users may run commands, but they cannot use any language elements.

CONSTRAINED LANGUAGE (Constrained Language)

The ConstrainedLanguage language mode permits all Windows cmdlets and all Windows PowerShell language elements, but it limits permitted types.

ConstrainedLanguage language mode is designed to support User Mode Code Integrity (UMCI) on Windows RT. It is the only supported language mode on Windows RT, but it is available on all supported systems.

UMCI protects ARM devices by allowing only Microsoft-signed and Microsoft-certified apps to be installed on Windows RT-based devices. ConstrainedLanguage mode prevents users from using Windows PowerShell to circumvent or violate UMCI.

The features of ConstrainedLanguage mode are as follows:

-- All cmdlets in Windows modules, and other UMCI-approved cmdlets, are fully functional and have complete access to system resources, except as noted.

-- All elements of the Windows PowerShell scripting language are permitted.

-- All modules included in Windows can be imported and all commands that the modules export run in the session.

-- In Windows PowerShell Workflow, you can write and run script workflows (workflows written in the Windows PowerShell language). XAML-based workflows are not supported and you cannot run XAML in a script workflow, such as by using "Invoke-Expression -Language XAML". Also, workflows cannot call other workflows, although nested workflows are permitted.

-- The Add-Type cmdlet can load signed assemblies, but it cannot load arbitrary C# code or Win32 APIs.

-- The New-Object cmdlet can be used only on allowed types (listed below).

-- Only allowed types (listed below) can be used in Windows PowerShell. Other types are not permitted.

-- Type conversion is permitted, but only when the result is an allowed type.

-- Cmdlet parameters that convert string input to types work only when the resulting type is an allowed type.

-- The ToString() method and the .NET methods of allowed types (listed below) can be invoked. Other methods cannot be invoked.

-- Users can get all properties of allowed types. Users can set the values of properties only on Core types.

-- Only the following COM objects are permitted.
  -- Scripting.Dictionary
  -- Scripting.FileSystemObject
  -- VBScript.RegExp

Allowed Types:

The following types are permitted in ConstrainedLanguage language mode. Users can get properties, invoke methods, and convert objects to these types.

AliasAttribute
AllowEmptyCollectionAttribute
AllowEmptyStringAttribute
AllowNullAttribute
Array
Bool
byte
char
CmdletBindingAttribute
DateTime
decimal
DirectoryEntry
DirectorySearcher
double
float
Guid
Hashtable
int
Int16

long
ManagementClass
ManagementObject
ManagementObjectSearcher
NullString
OutputTypeAttribute
ParameterAttribute
PSCredential
PSDefaultValueAttribute
PSListModifier
PSObject
PSPrimitiveDictionary
PSReference
PSTypeNameAttribute
Regex
SByte
string
SupportsWildcardsAttribute
SwitchParameter
System.Globalization.CultureInfo
System.Net.IPAddress
System.Net.Mail.MailAddress
System.Numerics.BigInteger
System.Security.SecureString
TimeSpan
UInt16
UInt32
UInt64

FINDING THE LANGUAGE MODE OF A SESSION CONFIGURATION

When a session configuration is created by using a session configuration file, the session configuration has a LanguageMode property. You can find the language mode by getting the value of the LanguageMode property.

```
PS C:\>(Get-PSSessionConfiguration -Name Test).LanguageMode
FullLanguage
```

On other session configurations, you can find the language mode indirectly by finding the language mode of a session that is created by using the session configuration.

FINDING THE LANGUAGE MODE OF A SESSION

You can find the language mode of a FullLanguage or ConstrainedLanguage session by getting the value of the LanguageMode property of the session state.

For example:

```
PS C:\>$ExecutionContext.SessionState.LanguageMode
ConstrainedLanguage
```

However, in sessions with RestrictedLanguage and NoLanguage language modes, you cannot use the dot method to get property values. Instead, the error message reveals the language mode.

When you run the $ExecutionContext.SessionState.LanguageMode command in a RestrictedLanguage session, Windows PowerShell returns the PropertyReferenceNotSupportedInDataSection and VariableReferenceNotSupportedInDataSection error messages.

```
PropertyReferenceNotSupportedInDataSection:
Property references are not allowed in restricted language
mode or a Data section.

VariableReferenceNotSupportedInDataSection
A variable that cannot be referenced in restricted language
mode or a Data section is being referenced.
```

When you run the $ExecutionContext.SessionState.LanguageMode command in a NoLanguage session, Windows PowerShell returns the ScriptsNotAllowed error message.

```
ScriptsNotAllowed
The syntax is not supported by this runspace. This might
be because it is in no-language mode.
```

KEYWORDS

- about_ConstrainedLanguage
- about_FullLanguage
- about_NoLanguage
- about_RestrictedLanguage

SEE ALSO

- about_Session_ConfigurationFiles
- about_Session_Configurations
- about_Windows_RT

TOPIC

about_Line_Editing

SHORT DESCRIPTION

Describes how to edit commands at the Windows PowerShell command prompt.

LONG DESCRIPTION

The Windows PowerShell console has some useful features to help you to edit commands at the Windows PowerShell command prompt.

Move Left and Right

To move the cursor one character to the left, press the LEFT ARROW key. To move the cursor one word to the left, press CTRL+LEFT ARROW. To move the cursor one character to the right, press the RIGHT ARROW key. To move the cursor one word to the right, press CTRL+RIGHT ARROW.

Line Start and End

To move to the beginning of a line, press the HOME key. To move to the end of a line, press the END key.

Delete Characters

To delete the character in behind the cursor, press the BACKSPACE key. To delete the character in front of the cursor, press the DELETE key.

Delete the Remainder of a Line

To delete all the characters in the line after the cursor, press CTRL+END.

Insert/Overstrike Mode

To change to overstrike mode, press the INSERT key. To return to insert mode, press INSERT again.

Tab Completion

To complete a command, such as the name of a cmdlet, a cmdlet parameter, or a path, press the TAB key. If the first suggestion that is displayed is not what you want, press the TAB key again.

SEE ALSO

about_Command_Syntax
about_Path_Syntax

TOPIC

about_Locations

SHORT DESCRIPTION

Describes how to access items from the working location in Windows PowerShell.

LONG DESCRIPTION

The current working location is the default location to which commands point. In other words, this is the location that Windows PowerShell uses if you do not supply an explicit path to the item or location that is affected by the command. In most cases, the current working location is a drive accessed through the Windows PowerShell FileSystem provider and, in some cases, a directory on that drive. For example, you might set your current working location to the following location:

C:\Program Files\Windows PowerShell

As a result, all commands are processed from this location unless

another path is explicitly provided.

Windows PowerShell maintains the current working location for each drive even when the drive is not the current drive. This allows you to access items from the current working location by referring only to the drive of another location. For example, suppose that your current working location is C:\Windows. Now, suppose you use the following command to change your current working location to the HKLM: drive:

```
Set-Location HKLM:
```

Although your current location is now the registry drive, you can still access items in the C:\Windows directory simply by using the C: drive, as shown in the following example:

```
Get-ChildItem C:
```

Windows PowerShell remembers that your current working location for that drive is the Windows directory, so it retrieves items from that directory. The results would be the same if you ran the following command:

```
Get-ChildItem C:\Windows
```

In Windows PowerShell, you can use the Get-Location command to determine the current working location, and you can use the Set-Location command to set the current working location. For example, the following command sets the current working location to the Windows directory of the C: drive:

```
Set-Location c:\windows
```

After you set the current working location, you can still access items from other drives simply by including the drive name (followed by a colon) in the command, as shown in the following example:

```
Get-ChildItem HKLM :\software
```

The example command retrieves a list of items in the Software container

of the HKEY Local Machine hive in the registry.

Windows PowerShell also allows you to use special characters to represent the current working location and its parent location. To represent the current working location, use a single period. To represent the parent of the current working location, use two periods. For example, the following specifies the System subdirectory in the current working location:

```
Get-ChildItem .\system
```

If the current working location is C:\Windows, this command returns a list of all the items in C:\Windows\System. However, if you use two periods, the parent directory of the current working directory is used, as shown in the following example:

```
Get-ChildItem ..\"program files"
```

In this case, Windows PowerShell treats the two periods as the C: drive, so the command retrieves all the items in the C:\Program Files directory.

A path beginning with a slash identifies a path from the root of the current drive. For example, if your current working location is C:\Program Files\Windows PowerShell, the root of your drive is C. Therefore, the following command lists all items in the C:\Windows directory:

```
Get-ChildItem \windows
```

If you do not specify a path beginning with a drive name, slash, or period when supplying the name of a container or item, the container or item is assumed to be located in the current working location. For example, if your current working location is C:\Windows, the following command returns all the items in the C:\Windows\System directory:

```
Get-ChildItem system
```

If you specify a file name rather than a directory name, Windows

PowerShell returns details about that file (assuming that file is located in the current working location).

SEE ALSO

Set-Location
about_Providers
about_Path_Syntax

TOPIC

about_Logical_Operators

SHORT DESCRIPTION

Describes the operators that connect statements in Windows PowerShell.

LONG DESCRIPTION

The Windows PowerShell logical operators connect expressions and statements, allowing you to use a single expression to test for multiple conditions.

For example, the following statement uses the and operator and the or operator to connect three conditional statements. The statement is true only when the value of $a is greater than the value of $b, and either $a or $b is less than 20.

```
($a -gt $b) -and (($a -lt 20) -or ($b -lt 20))
```

Windows PowerShell supports the following logical operators.

Operator	Description	Example
-and	Logical and. TRUE only when both statements are TRUE.	(1 -eq 1) -and (1 -eq 2) False
-or	Logical or. TRUE when either or both statements are TRUE.	(1 -eq 1) -or (1 -eq 2) True
-xor	Logical exclusive or. TRUE only when one of the statements is TRUE and the other is FALSE.	(1 -eq 1) -xor (2 -eq 2) False
-not	Logical not. Negates the statement that follows it.	-not (1 -eq 1) False
!	Logical not. Negates the statement that follows it. (Same as -not)	!(1 -eq 1) False

Note: The previous examples also use the equal to comparison
operator (-eq). For more information, see about_Comparison_Operators.
The examples also use the Boolean values of integers. The integer 0
has a value of FALSE. All other integers have a value of TRUE.

The syntax of the logical operators is as follows:

```
<statement> {-AND | -OR | -XOR} <statement>
{! | -NOT} <statement>
```

Statements that use the logical operators return Boolean (TRUE or FALSE)
values.

The Windows PowerShell logical operators evaluate only the statements
required to determine the truth value of the statement. If the left operand
in a statement that contains the and operator is FALSE, the right operand
is not evaluated. If the left operand in a statement that contains
the or statement is TRUE, the right operand is not evaluated. As a result,
you can use these statements in the same way that you would use
the If statement.

SEE ALSO
about_Operators
Compare-Object
about_Comparison_operators
about_If

TOPIC

about_methods

SHORT DESCRIPTION

Describes how to use methods to perform actions on objects in Windows PowerShell.

LONG DESCRIPTION

Windows PowerShell uses objects to represent the items in data stores or the state of the computer. For example, FileInfo objects represent the files in file system drives and ProcessInfo objects represent the processes on the computer.

Objects have properties, which store data about the object, and methods that let you change the object.

A "method" is a set of instructions that specify an action you can perform on the object. For example, the FileInfo object includes the CopyTo method that copies the file that the FileInfo object represents.

To get the methods of any object, use the Get-Member cmdlet. Use its MemberType property with a value of "Method". The following command gets the methods of process objects.

```
PS C:\>Get-Process | Get-Member -MemberType Method
```

```
   TypeName: System.Diagnostics.Process

Name                 MemberType Definition
----                 ---------- ----------
BeginErrorReadLine   Method     System.Void BeginErrorReadLine()
BeginOutputReadLine  Method     System.Void BeginOutputReadLine()
...
Kill                 Method     System.Void Kill()
Refresh              Method     System.Void Refresh()
Start                Method     bool Start()
ToString             Method     string ToString()
WaitForExit          Method     bool WaitForExit(int milliseconds), System.Void WaitForExit()
WaitForInputIdle     Method     bool WaitForInputIdle(int milliseconds), bool WaitForInputIdle()
```

To perform or "invoke" a method of an object, type a dot (.), the method name, and a set of parentheses "()". If the method has arguments, place the argument values inside the parentheses. The parentheses are required for every method call, even when there are no arguments.

For example, the following command invokes the Kill method of processes to end the Notepad process on the computer. As a result, the instance of Notepad closes. (The Get-Process command is enclosed in parentheses to insure that it runs before the Kill method is invoked.

```
(Get-Process Notepad).Kill()
```

Another very useful process is the Split method of strings. The split method takes a delimiter character argument that tells the method where to split the string.

```
PS C:\>$a = "Try-Catch-Finally"
PS C:\>$a.Split("-")
Try
Catch
Finally
```

As shown in the previous examples, you can invoke a method on an object that you get by using a command or an object in a variable.

Starting in Windows PowerShell 4.0, method invocation by using dynamic method names is supported.

LEARNING ABOUT METHODS

To find definitions of the methods of an object, go to help topic for for the object type in MSDN and look for its methods page. For example,

the following page describes the methods of process objects (System.Diagnostics.Process).

http://msdn.microsoft.com/library/system.diagnostics.process_methods

To determine the arguments of a method, review the method definition, which is like the syntax diagram of a Windows PowerShell cmdlet.

A method definition might have one or more method signatures, which are like the parameter sets of Windows PowerShell cmdlets. The signatures show all of the valid formats of commands to invoke the method.

For example, the CopyTo method of the FileInfo class contains the following two method signatures:

1. CopyTo(String destFileName)
2. CopyTo(String destFileName, Boolean overwrite)

The first method signature takes the destination file name (and a path). The following example use The first CopyTo method to copy the Final.txt file to the C:\Bin directory.

```
(Get-ChildItem c:\final.txt).CopyTo("c:\bin\final.txt")
```

The second method signature take a destination file name and a Boolean value that determines whether the destination file should be overwritten, if it already exists.

The following example use The second CopyTo method to copy the Final.txt file to the C:\Bin directory, and to overwrite existing files.

```
(Get-ChildItem c:\final.txt).CopyTo("c:\bin\final.txt", $true)
```

METHODS OF SCALAR OBJECTS AND COLLECTIONS

The methods of one ("scalar") object of a particular type are often different from the methods of a collection of objects of the same type.

For example, every process has a Kill method, but a collection of processes does not have a Kill method.

Beginning in Windows PowerShell 3.0, Windows PowerShell tries to prevent scripting errors that result from the differing methods of scalar objects and collections.

Beginning in Windows PowerShell 4.0, collection filtering by using a method syntax is supported.

If you submit a collection, but request a method that exists only on single ("scalar") objects, Windows invokes the method on every object in the collection.

If the method exists on the individual objects and on the collection, Windows PowerShell does not alter the result.

This feature also works on properties of scalar objects and collections. For more information, see about_Properties.

EXAMPLES

The following example runs the Kill method of individual process objects on a collection of process objects. This example works only on Windows PowerShell 3.0 and later versions of Windows PowerShell.

The first command starts three instances of the Notepad process. The second command uses the Get-Process command to get all three instance of the Notepad process and save them in the $p variable.

```
PS C:\>Notepad; Notepad; Notepad
PS C:\>$p = Get-Process Notepad
```

The third command uses the Count property of all collections to verify that there are three processes in the $p variable.

```
PS C:\>$p.Count
3
```

The fourth command runs the Kill method on all three processes in the $p variable.

This command works even though a collection of processes does not have a Kill method.

```
PS C:\>$p.Kill()
```

The fifth command uses the Get-Process command to confirm that the Kill command worked.

```
PS C:\>Get-Process Notepad
Get-Process : Cannot find a process with the name "notepad". Verify the process name and call the cmdlet again.
At line:1 char:12
```

```
    + get-process <<<<  notepad
      + CategoryInfo          : ObjectNotFound: (notepad:String) [Get-Process],
ProcessCommandException
      + FullyQualifiedErrorId :
NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.GetProcessCommand
```

To perform the same task on Windows PowerShell 2.0,
use the Foreach-Object cmdlet to run the method
on each object in the collection.

```
    PS C:\>$p | Foreach-Object {$_.Kill()}
```

SEE ALSO

about_Objects
about_Properties
Get-Member

TOPIC

about_Modules

SHORT DESCRIPTION

Explains how to install, import, and use Windows PowerShell modules.

LONG DESCRIPTION

A module is a package that contains Windows PowerShell commands, such as cmdlets, providers, functions, workflows, variables, and aliases.

People who write commands can use modules to organize their commands and share them with others. People who receive modules can add the commands in the modules to their Windows PowerShell sessions and use them just like the built-in commands.

This topic explains how to use Windows PowerShell modules. For information

about how to write Windows PowerShell modules, see "Writing a Windows PowerShell Module" in the MSDN (Microsoft Developer Network) library at http://go.microsoft.com/fwlink/?LinkId=144916.

WHAT IS A MODULE?

A module is a package of commands. All cmdlets and providers in your session are added by a module or a snap-in.

WHAT'S NEW IN MODULES: Module Auto-Loading

Beginning in Windows PowerShell 3.0, Windows PowerShell imports modules automatically the first time that you run any command in an installed module. You can now use the commands in a module without any set-up or profile configuration, so there's no need to manage modules after you install them on your computer.

The commands in a module are also easier to find. The Get-Command cmdlet now gets all commands in all installed modules, even if they are not yet in the session, so you can find a command and use it without importing.

Any of the following commands will import a module into your session.

```
#Run the command
Get-Mailbox –Identity Chris

#Get the command
Get-Command Get-Mailbox

#Get help for the command
Get-Help Get-Mailbox
```

Get-Command commands that include a wildcard character (*) are considered to be for discovery, not use, and do not import any modules.

Only modules that are stored in the location specified by the PSModulePath environment variable are automatically imported. Modules in other locations must be imported by running the Import-Module cmdlet.

Also, commands that use Windows PowerShell providers do not automatically import a module. For example, if you use a command that requires the WSMan: drive, such as the Get-PSSessionConfiguration cmdlet, you might need to run the Import-Module cmdlet to import the Microsoft.WSMan.Management module that includes the WSMan: drive.

You can still run the Import-Module command to import a module and use the $PSModuleAutoloadingPreference variable to enable, disable and configure automatic importing of modules. For more information, see about_Preference_Variables.

HOW TO USE A MODULE

To use a module, perform the following tasks:

1. Install the module. (This is often done for you.)
2. Find the commands that the module added.
3. Use the commands that the module added.

This topic explains how to perform these tasks. It also includes other useful information about managing modules.

HOW TO INSTALL A MODULE

If you receive a module as a folder with files in it, you need to install it on your computer before you can use it in Windows PowerShell.

Most modules are installed for you. Windows PowerShell comes with several preinstalled modules, sometimes called the "core" modules. On Windows-based computers, if features that are included with the operating system have cmdlets to manage them, those modules are preinstalled. When you install a Windows feature, by using, for example, the Add Roles and Features Wizard in Server Manager, or the Turn Windows features on or off dialog box in Control Panel, any Windows PowerShell modules that are part of the feature are installed. Many other modules come in an installer or Setup program that installs the module.

To install a module folder:

1. Create a Modules directory for the current user if one does not exist.

   To create a Modules directory, type:

   ```
   New-Item -Type Directory -Path $home\Documents\WindowsPowerShell\Modules
   ```

2. Copy the entire module folder into the Modules directory.

   You can use any method to copy the folder, including Windows Explorer and Cmd.exe, as well as Windows PowerShell.

   In Windows PowerShell use the Copy-Item cmdlet. For example, to copy the MyModule folder from C:\ps-test\MyModule to the Modules directory, type:

   ```
   Copy-Item -Path c:\ps-test\MyModule -Destination
$home\Documents\WindowsPowerShell\Modules
   ```

You can install a module in any location, but installing your modules in a default module location makes them easier to manage. For more information about

the default module locations, see the "MODULE AND DSC RESOURCE LOCATIONS, AND PSMODULEPATH" section.

HOW TO FIND INSTALLED MODULES

To find modules that are installed in a default module location, but not yet imported into your session, type:

```
Get-Module -ListAvailable
```

To find the modules that have already been imported into your session, at the Windows PowerShell prompt, type:

```
Get-Module
```

For more information about the Get-Module cmdlet, see Get-Module.

HOW TO FIND THE COMMANDS IN A MODULE

Use the Get-Command cmdlet to find all available commands. You can use the parameters of the Get-Command cmdlet to filter commands such as by module, name, and noun.

To find all commands in a module, type:

```
Get-Command -Module <module-name>
```

For example, to find the commands in the BitsTransfer module, type:

```
Get-Command -Module BitsTransfer
```

For more information about the Get-Command cmdlet, see Get-Command.

HOW TO GET HELP FOR THE COMMANDS IN A MODULE

If the module contains Help files for the commands that it exports, the Get-Help cmdlet will display the Help topics. Use the same Get-Help command format that you would use to get help for any command in Windows PowerShell.

Beginning in Windows PowerShell 3.0, you can download Help files for a module and download updates to the Help files so they are never obsolete.

To get help for a commands in a module,

type:

Get-Help <command-name>

To get help online for command in a module, type:
Get-Help <command-name> -Online

To download and install the help files for the commands in a module,
type:
Update-Help –Module <module-name>

For more information, see Get-Help and Update-Help.

HOW TO IMPORT A MODULE

You might have to import a module or import a module file. Importing is required when a module is not installed in the locations specified by the PSModulePath environment variable ($env:PSModulePath), or the module consists of file, such as a .dll or .psm1 file, instead of typical module that is delivered as a folder.

You might also choose to import a module so that you can use the parameters of the Import-Module command, such as the Prefix parameter, which adds a distinctive prefix to the noun names of all imported commands, or the NoClobber parameter, which prevents the module from adding commands that would hide or replace existing commands in the session.

To import modules, use the Import-Module cmdlet.

To import modules in a PSModulePath location into the current session, use the following command format.

Import-Module <module-name>

For example, the following command imports the BitsTransfer module into the current session.

Import-Module BitsTransfer

To import a module that is not in a default module location, use the fully qualified path to the module folder in the command.

For example, to add the TestCmdlets module in the C:\ps-test directory to your session, type:

Import-Module c:\ps-test\TestCmdlets

To import a module file that is not contained in a module folder, use the fully qualified path to the module file in the command.

For example, to add the TestCmdlets.dll module in the C:\ps-test directory to your session, type:

```
Import-Module c:\ps-test\TestCmdlets.dll
```

For more information about adding modules to your session, see Import-Module.

HOW TO IMPORT A MODULE INTO EVERY SESSION

The Import-Module command imports modules into your current Windows PowerShell session. This command affects only the current session.

To import a module into every Windows PowerShell session that you start, add the Import-Module command to your Windows PowerShell profile.

For more information about profiles, see about_Profiles.

HOW TO REMOVE A MODULE

When you remove a module, the commands that the module added are deleted from the session.

To remove a module from your session, use the following command format.

```
remove-module <module-name>
```

For example, the following command removes the BitsTransfer module from the current session.

```
remove-module BitsTransfer
```

Removing a module reverses the operation of importing a module. Removing a module does not uninstall the module. For more information about the Remove-Module cmdlet, see Remove-Module.

MODULE AND DSC RESOURCE LOCATIONS, AND PSMODULEPATH

The following are default locations for Windows PowerShell modules. Starting in Windows PowerShell 4.0, with the introduction of DSC, a new default module and DSC resource folder was introduced. For more

information about DSC, see about_DesiredStateConfiguration.

System: $pshome\Modules
(%windir%\System32\WindowsPowerShell\v1.0\Modules)
System modules are those that ship with Windows and Windows PowerShell.

Starting in Windows Powershell 4.0, when Windows PowerShell Desired State Configuration (DSC) was introduced, DSC resources that are included with Windows PowerShell are also stored in $pshome\Modules, in the $pshome\Modules\PSDesiredStateConfiguration\DSCResources folder.

Current user: $home\Documents\WindowsPowerShell\Modules
(%UserProfile%\Documents\WindowsPowerShell\Modules)

- or -

$home\My Documents\WindowsPowerShell\Modules
(%UserProfile%\My Documents\WindowsPowerShell\Modules)
This is the location for user-added modules prior to Windows PowerShell 4.0.

In Windows PowerShell 4.0 and later releases of Windows PowerShell, user-added modules and DSC resources are stored in C:\Program Files\WindowsPowerShell\Modules. Modules and DSC resources in this location are accessible by all users of the computer.
This change was required because the DSC engine runs as local system, and could not access user-specific paths, such as $home\Documents\WindowsPowerShell\Modules.

Starting in Windows PowerShell 5.0, with the addition of the PowerShellGet module, and the PowerShell Gallery of community- and Microsoft-created resources (https://www.powershellgallery.com), the Install-Module command installs modules and DSC resources to C:\Program Files\WindowsPowerShell\Modules by default.

Note: To add or change files in the %Windir%\System32 directory, start Windows PowerShell with the "Run as administrator" option.

You can change the default module locations on your system by changing the value of the PSModulePath environment variable ($Env:PSModulePath). The PSModulePath environment variable is modeled on the Path environment variable and has the same format.

To view the default module locations, type:
$env:psmodulepath

To add a default module location, use the following command format.

$env:psmodulepath = $env:psmodulepath + ";<path>"

The semi-colon (;) in the command separates the new path from the path that precedes it in the list.

For example, to add the "C:\ps-test\Modules" directory, type:
$env:psmodulepath + ";c:\ps-test\Modules"

When you add a path to PSModulePath, Get-Module and Import-Module commands include modules in that path.

The value that you set affects only the current session. To make the change persistent, add the command to your Windows PowerShell profile or use System in Control Panel to change the value of the PSModulePath environment variable in the registry.

Also, to make the change persistent, you can also use the SetEnvironmentVariable method of the System.Environment class to add a Path to the PSModulePath environment variable.

For more information about the PSModulePath variable, see about_Environment_Variables.

MODULES AND NAME CONFLICTS

Name conflicts occur when more than one command in the session has the same name. Importing a module causes a name conflict when commands in the module have the same names as commands or items in the session.

Name conflicts can result in commands being hidden or replaced.

-- Hidden. A command is hidden when it is not the command that runs when you type the command name, but you can run it by using another method, such as by qualifying the command name with the name of the module or snap-in in which it originated.

-- Replaced. A command is replaced when you cannot run it because it has been overwritten by a command with the same name. Even when you remove the module that caused the conflict, you cannot run a replaced command unless you restart the session.

Import-Module might add commands that hide and replace commands in the current session. Also, commands in your session can hide commands that the module added.

To detect name conflicts, use the All parameter of the Get-Command cmdlet. Beginning in Windows PowerShell 3.0, Get-Command gets only that commands that run when you type the command name. The All parameter gets all commands with the specific name in the session.

To prevent name conflicts, use the NoClobber or Prefix parameters of the Import-Module cmdlet. The Prefix parameter adds a prefix to the names of imported commands so that they are unique in the session. The NoClobber parameter does not import any commands that would hide or replace existing commands in the session.

You can also use the Alias, Cmdlet, Function, and Variable parameters of Import-Module to select only the commands that you want to import, and you can exclude commands that cause name conflicts in your session.

Module authors can prevent name conflicts by using the DefaultCommandPrefix property of the module manifest to add a default prefix to all command names. The value of the Prefix parameter takes precedence over the value of DefaultCommandPrefix.

Even if a command is hidden, you can run it by qualifying the command name with the name of the module or snap-in in which it originated.

The Windows PowerShell command precedence rules determine which command runs when the session includes commands with the same name.

For example, when a session includes a function and a cmdlet with the same name, Windows PowerShell runs the function by default. When the session includes commands of the same type with the same name, such as two cmdlets with the same name, by default, it runs the most recently added command.

For more information, including an explanation of the precedence rules and instructions for running hidden commands, see about_Command_Precedence.

MODULES AND SNAP-INS

You can add commands to your session from modules and snap-ins. Modules can add all types of commands, including cmdlets, providers, and functions, and items, such as variables, aliases, and Windows PowerShell drives. Snap-ins can add only cmdlets and providers.

Before removing a module or snap-in from your session, use the following commands to determine which commands will be removed.

To find the source of a cmdlet in your session, use the following command format:

get-command <cmdlet-name> | format-list -property verb, noun, pssnapin, module

For example, to find the source of the Get-Date cmdlet, type:

get-command get-date | format-list -property verb, noun, pssnapin, module

For more information about Windows PowerShell snap-ins, see about_PSSnapins.

MODULE-RELATED WARNINGS AND ERRORS

The commands that a module exports should follow the Windows PowerShell command naming rules. If the module that you import exports cmdlets or functions that have unapproved verbs in their names, the Import-Module cmdlet displays the following warning message.

WARNING: Some imported command names include unapproved verbs which might make them less discoverable. Use the Verbose parameter for more detail or type Get-Verb to see the list of approved verbs.

This message is only a warning. The complete module is still imported, including the non-conforming commands. Although the message is displayed to module users, the naming problem should be fixed by the module author.

To suppress the warning message, use the DisableNameChecking parameter of the Import-Module cmdlet.

BUILT-IN MODULES AND SNAP-INS

In Windows PowerShell 2.0 and in older-style host programs in Windows PowerShell 3.0 and later, the core commands that are installed with Windows PowerShell are packaged in snap-ins that are added automatically to every Windows PowerShell session.

Beginning in Windows PowerShell 3.0, in newer-style host programs -- those that implement the InitialSessionState.CreateDefault2 initial session state API -- the core commands are packaged in modules. The default is Microsoft.PowerShell.Core, which is always a snap-in.

The Microsoft.PowerShell.Core snap-in is added to every session by default. Modules are loaded automatically on first-use.

NOTE: Remote sessions, including sessions that are started by using the New-PSSession cmdlet, are older-style sessions in which the built-in commands are packaged in snap-ins.

The following modules (or snap-ins) are installed with Windows PowerShell.

Microsoft.PowerShell.Archive

Microsoft.PowerShell.Core
Microsoft.PowerShell.Diagnostics
Microsoft.PowerShell.Host
Microsoft.PowerShell.Management
Microsoft.PowerShell.ODataUtils
Microsoft.PowerShell.Security
Microsoft.PowerShell.Utility
Microsoft.WSMan.Management
OneGet
PowerShellGet
PSDesiredStateConfiguration
PSScheduledJob
PSWorkflow
PSWorkflowUtility
ISE

LOGGING MODULE EVENTS

Beginning in Windows PowerShell 3.0, you can record execution events for the cmdlets and functions in Windows PowerShell modules and snap-ins by setting the LogPipelineExecutionDetails property of modules and snap-ins to $True. You can also use a Group Policy setting, Turn on Module Logging, to enable module logging in all Windows PowerShell sessions. For more information, see about_EventLogs (http://go.microsoft.com/fwlink/?LinkID=113224) and about_Group_Policy_Settings (http://go.microsoft.com/fwlink/?LinkID=251696).

SEE ALSO

about_Command_Precedence
about_DesiredStateConfiguration
about_EventLogs
about_Group_Policy_Settings
about_PSSnapins
Get-Command
Get-Help
Get-Module
Import-Module
Remove-Module

TOPIC

about_Objects

SHORT DESCRIPTION

Provides essential information about objects in Windows PowerShell.

LONG DESCRIPTION

Every action you take in Windows PowerShell occurs within the context of objects. As data moves from one command to the next, it moves as one or more identifiable objects. An object, then, is a collection of data that represents an item. An object is made up of three types of data: the objects type, its methods, and its properties.

TYPES, PROPERTIES, AND METHODS

The object type tells what kind of object it is. For example, an object that represents a file is a FileInfo object.

The object methods are actions that you can perform on the object. For example, FileInfo objects have a CopyTo method that you can use to copy the file.

Object properties store information about the object. For example, FileInfo objects have a LastWriteTime property that stores the date and time that the file was most recently accessed.

When working with objects, you can use their properties and methods in commands to take action and manage data.

OBJECTS IN PIPELINES

When commands are combined in a pipeline, they pass information to each other as objects. When the first command runs, it sends one or more objects down the pipeline to the second command. The second command receives the objects from the first command, processes the objects, and then passes new or revised objects to the next command in the pipeline. This continues until all commands in the pipeline run.

The following example demonstrates how objects are passed from one command to the next:

```
Get-ChildItem C: | where {$_.PsIsContainer -eq $False} |
Format-List
```

The first command (Get-ChildItem C:) returns a file or directory object for each item in the root directory of the file system. The file and directory objects are passed down the pipeline to the second command.

The second command (where {$_.PsIsContainer -eq $false}) uses the PsIsContainer property of all file system objects to select only files, which have a value of False ($false) in their PsIsContainer property. Folders, which are containers and, thus, have a value of True ($true) in their PsIsContainer property, are not selected.

The second command passes only the file objects to the third command (Format-List), which displays the file objects in a list.

FOR MORE INFORMATION

Now that you understand a bit about objects, see the about_Methods help topic to learn how to find and use object methods, the about_Properties topic to learn how to find and use object properties, and the Get-Member topic, to learn how to find an object type.

SEE ALSO

about_Methods
about_Object_Creation
about_Properties
about_Pipelines
Get-Member

TOPIC

about_Object_Creation

SHORT DESCRIPTION

Explains how to create objects in Windows PowerShell.

LONG DESCRIPTION

You can create objects in Windows PowerShell and use the objects that you create in commands and scripts.

There are several ways to create objects:

-- New-Object: The New-Object cmdlet creates an instance of a .NET Framework object or COM object.

-- Hash tables: Beginning in Windows PowerShell 3.0, you can create objects from hash tables of property names and property values.

-- Import-Csv: The Import-Csv cmdlet creates custom objects (PSCustomObject) from the items in a CSV file. Each row is an object instance and each column is an object property.

This topic will demonstrate and discuss each of these methods.

NEW-OBJECT

The New-Object cmdlet provides a robust and consistent way to create new objects. The cmdlet works with almost all types and in all supported versions of Windows PowerShell.

To create a new object, specify either the type of a .NET Framework class or a ProgID of a COM object.

For example, the following command creates a Version object.

```
PS C:\>$v = New-Object -TypeName System.Version -ArgumentList 2.0.0.1
PS C:\>$v

Major  Minor  Build  Revision
-----  -----  -----  --------
2      0      0      1

PS C:\>$v | Get-Member

   TypeName: System.Version
   ...
```

For more information, see the help topic for the New-Object cmdlet.

CREATE OBJECTS FROM HASH TABLES

Beginning in Windows PowerShell 3.0, you can create an object from a hash table of properties and property values.

The syntax is as follows:

```
[<class-name>]@{<property-name>=<property-value>;<property-name>=<property-value>}
```

This method works only for classes that have a null constructor, that is, a constructor that has no parameters. The object properties must be public and settable.

CREATE CUSTOM OBJECTS FROM HASH TABLES

Custom objects are very useful and they are very easy to create by using the hash table method. To create a custom object, use the PSCustomObject class, a class designed specifically for this purpose.

Custom objects are an excellent way to return customized output from a function or script; far more useful than returning formatted output that cannot be reformatted or piped to other commands.

The commands in the Test-Object function set some variable values and then use those values to create a custom object. (You can see this object in use in the example section of the Update-Help cmdlet help topic.)

```
function Test-Object
{   $ModuleName = "PSScheduledJob"
    $HelpCulture = "en-us"
    $HelpVersion = "3.1.0.0"
    [PSCustomObject]@{"ModuleName"=$ModuleName; "UICulture"=$HelpCulture;
"Version"=$HelpVersion}

    $ModuleName = "PSWorkflow"
    $HelpCulture = "en-us"
    $HelpVersion = "3.0.0.0"
    [PSCustomObject]@{"ModuleName"=$ModuleName; "UICulture"=$HelpCulture;
"Version"=$HelpVersion}
}
```

The output of this function is a collection of custom objects formatted as a table by default.

```
PS C:\>Test-Object

ModuleName       UICulture     Version
---------        ---------     -------
PSScheduledJob   en-us         3.1.0.0
PSWorkflow       en-us         3.0.0.0
```

Users can manage the properties of the custom objects just as they do with standard objects.

```
PS C:\>(Test-Object).ModuleName
PSScheduledJob
PSWorkflow
```

CREATE NON-CUSTOM OBJECTS FROM HASH TABLES

You can also use hash tables to create objects for non-custom classes. When you create an object for a non-custom class, the full namespace name is required unless class is in the System namespace. Use only the properties of the class.

For example, the following command creates a session option object.

```
[System.Management.Automation.Remoting.PSSessionOption]@{IdleTimeout=43200000;
SkipCnCheck=$True}
```

The requirements of the hash table feature, especially the null constructor requirement, eliminate many existing classes. However, most Windows PowerShell option classes are designed to work with this feature, as well as other very useful classes, such as the ScheduledJobTrigger class.

```
[Microsoft.PowerShell.ScheduledJob.ScheduledJobTrigger]@{Frequency="Daily";At="15:00"}

Id       Frequency      Time                DaysOfWeek          Enabled
--       ---------      ----                ----------          -------
0        Daily          6/6/2012 3:00:00 PM                     True
```

You can also use the hash table feature when setting parameter values. For example, the value of the SessionOption parameter of the New-PSSession cmdlet and the value of the JobTrigger parameter of Register-ScheduledJob can be a hash table.

```
New-PSSession -ComputerName Server01 -SessionOption @{IdleTimeout=43200000;
SkipCnCheck=$True}
```

Register-ScheduledJob Name Test -FilePath .\Get-Inventory.ps1 -Trigger @{Frequency="Daily";At="15:00"}

IMPORT-CSV

You can create custom objects from the items in a CSV file. When you use the Import-Csv cmdlet to import the CSV file, the cmdlet creates a custom object (PSCustomObject) for each item in the file. The column names are the object properties.

For example, if you import a CSV file of computer asset data, Import-CSV creates a collection of custom objects from the input.

```
#In Servers.csv
AssetID, Name, OS, Department
003, Server01, Windows Server 2012, IT
103, Server33, Windows 7, Marketing
212, Server35, Windows 8, Finance

PS C:\>$a = Import-Csv Servers.csv
PS C:\>$a

AssetID      Name         OS               Department
-------      ----         --               ----------
003          Server01     Windows Server 2012   IT
103          Server33     Windows 7          Marketing
212          Server35     Windows 8          Finance
```

Use the Get-Member cmdlet to confirm the object type.

```
PS C:\>$a | Get-Member

   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()
AssetID    NoteProperty System.String AssetID=003
Department NoteProperty System.String Department=IT
Name       NoteProperty System.String Name=Server01
OS         NoteProperty System.String OS=Windows Server 2012
```

You can use the custom objects just as you would standard

objects.

```
PS C:\>$a | where {$_.OS -eq "Windows 8"}

AssetID      Name         OS              Department
-------      ----         --              ----------
212          Server35     Windows 8       Finance
```

For more information, see the help topic for the Import-Csv cmdlet.

SEE ALSO

about_Objects
about_Methods
about_Properties
about_Pipelines
Get-Member
Import-Csv
New-Object

TOPIC

about_Operators

SHORT DESCRIPTION

Describes the operators that are supported by Windows PowerShell.

LONG DESCRIPTION

An operator is a language element that you can use in a command or expression. Windows PowerShell supports several types of operators to help you manipulate values.

Arithmetic Operators

Use arithmetic operators (+, -, *, /, %) to calculate values in a command

or expression. With these operators, you can add, subtract, multiply, or divide values, and calculate the remainder (modulus) of a division operation.

You can also use arithmetic operators with strings, arrays, and hash tables. The addition operator concatenates elements. The multiplication operator returns the specified number of copies of each element.

For more information, see about_Arithmetic_Operators.

Assignment Operators

Use assignment operators (=, +=, -=, *=, /=, %=) to assign one or more values to variables, to change the values in a variable, and to append values to variables. You can also cast the variable as any Microsoft .NET Framework data type, such as string or DateTime, or Process variable.

For more information, see about_Assignment_Operators.

Comparison Operators

Use comparison operators (-eq, -ne, -gt, -lt, -le, -ge) to compare values and test conditions. For example, you can compare two string values to determine whether they are equal.

The comparison operators include the match operators (-match, -notmatch), which find patterns by using regular expressions; the replace operator (-replace), which uses regular expressions to change input values; the like operators (-like, -notlike), which find patterns using wildcard characters (*); and the containment operators (-in, -notin, -contains, -notcontains), which determine whether a test value appears in a reference set.

They also include the bitwise operators (-bAND, -bOR, -bXOR, -bNOT) to manipulate the bit patterns in values.

For more information, see about_Comparison_Operators

Logical Operators

Use logical operators (-and, -or, -xor, -not, !) to connect conditional statements into a single complex conditional. For example, you can use a logical -and operator to create an object filter with two different

conditions.

For more information, see about_Logical_Operators.

Redirection Operators

Use redirection operators (>, >>, 2>, 2>, and 2>&1) to send the output of a command or expression to a text file. The redirection operators work like the Out-File cmdlet (without parameters) but they also let you redirect error output to specified files. You can also use the Tee-Object cmdlet to redirect output.

For more information, see about_Redirection.

Split and Join Operators

The -split and -join operators divide and combine substrings. The -split operator splits a string into substrings. The -join operator concatenates multiple strings into a single string.

For more information, see about_Split and about_Join.

Type Operators

Use the type operators (-is, -isnot, -as) to find or change the .NET Framework type of an object.

For more information, see about_Type_Operators.

Unary Operators

Use unary operators to increment or decrement variables or object properties and to set integers to positive or negative numbers. For example, to increment the variable $a from 9 to 10, you type $a++.

Special Operators

Use special operators to perform tasks that cannot be performed by the other types of operators. For example, special operators allow you to perform operations such as running commands and changing a value's data type.

@( ) Array subexpression operator

Returns the result of one or more statements as an array.
If there is only one item, the array has only one member.

```
@(Get-WMIObject win32_logicalDisk)
```

& Call operator

Runs a command, script, or script block. The call operator, also known as the "invocation operator," lets you run commands that are stored in variables and represented by strings. Because the call operator does not parse the command, it cannot interpret command parameters.

```
C:\PS> $c = "get-executionpolicy"
C:\PS> $c
get-executionpolicy

C:\PS> & $c
AllSigned
```

[ ] Cast operator

Converts or limits objects to the specified type. If the objects cannot be converted, Windows PowerShell generates an error.

```
[datetime]$birthday = "1/20/88"
[int64]$a = 34
```

, Comma operator

As a binary operator, the comma creates an array. As a unary operator, the comma creates an array with one member. Place the comma before the member.

```
$myArray = 1,2,3
$SingleArray = ,1
```

. Dot sourcing operator

Runs a script in the current scope so that any functions, aliases, and variables that the script creates are added to the current scope.

```
. c:\scripts.sample.ps1
```

Note: The dot sourcing operator is followed by a space. Use the space to distinguish the dot from the dot (.) symbol that represents the current directory.

In the following example, the Sample.ps1 script in the current directory is run in the current scope.

```
. .\sample.ps1
```

-f Format operator

Formats strings by using the format method of string objects. Enter the format string on the left side of the operator and the objects to be formatted on the right side of the operator.

```
C:\PS> "{0} {1,-10} {2:N}" -f 1,"hello",[math]::pi
1 hello      3.14
```

For more information, see the String.Format method (http://go.microsoft.com/fwlink/?LinkID=166450) and Composite Formatting (http://go.microsoft.com/fwlink/?LinkID=166451).

[ ] Index operator

Selects objects from indexed collections, such as arrays and hash tables. Array indexes are zero-based, so the first object is indexed as [0]. For arrays (only), you can also use negative indexes to get the last values. Hash tables are indexed by key value.

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

```
C:\PS> (get-hotfix | sort installedOn)[-1]
```

```
C:\PS> $h = @{key="value"; name="Windows PowerShell"; version="2.0"}
C:\PS> $h["name"]
```

Windows PowerShell

```
C:\PS> $x = [xml]"<doc><intro>Once upon a time...</intro></doc>"
C:\PS> $x["doc"]
intro
-----
Once upon a time...
```

| Pipeline operator

Sends ("pipes") the output of the command that precedes it to the command that follows it. When the output includes more than one object (a "collection"), the pipeline operator sends the objects one at a time.

```
get-process | get-member
get-pssnapin | where {$_.vendor -ne "Microsoft"}
```

. Property dereference operator

Accesses the properties and methods of an object.

```
$myProcess.peakWorkingSet
(get-process PowerShell).kill()
```

.. Range operator

Represents the sequential integers in an integer array, given an upper and lower boundary.

```
1..10
10..1
foreach ($a in 1..$max) {write-host $a}
```

:: Static member operator

Calls the static properties operator and methods of a .NET Framework class. To find the static properties and methods of an object, use the Static parameter of the Get-Member cmdlet.

[datetime]::now

$( ) Subexpression operator

Returns the result of one or more statements. For a single result, returns a scalar. For multiple results, returns an array.

$($x * 23)
$(Get-WMIObject win32_Directory)

SEE ALSO

about_Arithmetic_Operators
about_Assignment_Operators
about_Comparison_Operators
about_Logical_Operators
about_Type_Operators
about_Split
about_Join
about_Redirection

TOPIC

about_Operator_Precedence

SHORT DESCRIPTION

Lists the Windows PowerShell operators in precedence order.

[This topic was contributed by Kirk Munro, a Windows PowerShell MVP
from Ottawa, Ontario]

LONG DESCRIPTION

Windows PowerShell operators let you construct simple, but powerful expressions. This topic lists the operators in precedence order. Precedence order is the order in which Windows PowerShell evaluates the operators when multiple operators appear in the same expression.

When operators have equal precedence, Windows PowerShell evaluates them from left to right. The exceptions are the assignment operators, the cast operators, and the negation operators (!, -not, -bnot), which are evaluated from right to left.

You can use enclosures, such as parentheses, to override the standard precedence order and force Windows PowerShell to evaluate the enclosed part of an expression before an unenclosed part.

In the following list, operators are listed in the order that they are evaluated. Operators on the same line, or in the same group, have equal precedence.

The Operator column lists the operators. The Reference column lists the Windows PowerShell Help topic in which the operator is described. To display the topic, type "get-help <topic-name>".

OPERATOR	REFERENCE
$() @()	about_Operators
. (dereference) :: (static)	about_Operators
[0] (index operator)	about_Operators
[int] (cast operators)	about_Operators
-split (unary) -join (unary)	about_Split, about_Join
, (comma operator)	about_Operators
++ --	about_Assignment_Operators
-not ! -bNot	about_Logical_Operators, about_Comparison_Operators
.. (range operator)	about_Operators

-f (format operator) about_Operators

* / % about_Arithmetic_Operators

+ - about_Arithmetic_Operators

The following group of operators have equal precedence. Their case-sensitive and explicitly case-insensitive variants have the same precedence.

-split (binary) about_Split
-join (binary) about_Join
-is -isnot -as about_Type_Operators
-eq -ne -gt -gt -lt -le about_Comparison_Operators
-like -notlike about_comparison_operators
-match -notmatch about_comparison_operators
-in -notIn about_comparison_operators
-contains -notContains about_comparison_operators
-replace about_comparison_operators

The list resumes here with the following operators in precedence order:

-band -bor -bxor about_Comparison_Operators

-and -or -xor about_Comparison_Operators

. (dot-source) & (call) about_Scopes, about_Operators

| (pipeline operator) about_Operators

> >> 2> 2>> 2>&1 about_Redirection

= += -= *= /= %= about_Assignment_Operators

EXAMPLES

The following two commands show the arithmetic operators and the effect of using parentheses to force Windows PowerShell to evaluate the enclosed part of the expression first.

```
C:\PS> 2 + 3 * 4
14
```

```
C:\PS> (2 + 3) * 4
20
```

The following example gets the read-only text files from the local directory and saves them in the $read_only variable.

```
$read_only = get-childitem *.txt | where-object {$_.isReadOnly}
```

It is equivalent to the following example.

```
$read_only = ( get-childitem *.txt | where-object {$_.isReadOnly} )
```

Because the pipeline operator (|) has a higher precedence than the assignment operator (=), the files that the Get-ChildItem cmdlet gets are sent to the Where-Object cmdlet for filtering before they are assigned to the $read_only variable.

The following example demonstrates that the index operator takes precedence over the cast operator.

The first expression creates an array of three strings. Then, it uses the index operator with a value of 0 to select the first object in the array, which is the first string. Finally, it casts the selected object as a string. In this case, the cast has no effect.

```
C:\PS> [string]@('Windows','PowerShell','2.0')[0]
Windows
```

The second expression uses parentheses to force the cast operation to occur before the index selection. As a result, the entire array is cast as a (single) string. Then, the index operator selects the first item in the string array, which is the first character.

```
C:\PS> ([string]@('Windows','PowerShell','2.0'))[0]
W
```

In the following example, because the -gt (greater-than) operator has a higher precedence than the -and (logical AND) operator, the

result of the expression is FALSE.

```
C:\PS> 2 -gt 4 -and 1
False
```

It is equivalent to the following expression.

```
C:\PS> (2 -gt 4) -and 1
False
```

If the -and operator had higher precedence, the answer would be TRUE.

```
C:\PS> 2 -gt (4 -and 1)
True
```

However, this example demonstrates an important principle of managing operator precedence. When an expression is difficult for people to interpret, use parentheses to force the evaluation order, even when it forces the default operator precedence. The parentheses make your intentions clear to people who are reading and maintaining your scripts.

SEE ALSO

about_Assignment_Operators
about_Comparison_Operators
about_Join
about_Logical_Operators
about_Operators
about_Redirection
about_Scopes
about_Split
about_Type_Operators

TOPIC

about_PackageManagement

SHORT DESCRIPTION

PackageManagement is an aggregator for software package managers.

LONG DESCRIPTION

PackageManagement functionality was introduced in Windows PowerShell 5.0.

PackageManagement is a unified interface for software package management systems; you can run PackageManagement cmdlets to perform software discovery, installation, and inventory (SDII) tasks. Regardless of the underlying installation technology, you can run the common cmdlets in the PackageManagement module to search for, install, or uninstall packages; add, remove, and query package repositories; and run queries on a computer to determine which software packages are installed.

PackageManagement supports a flexible plug-in model that enables support for other software package management systems.

The PackageManagement module is included with Windows PowerShell 5.0 and later releases of Windows PowerShell, and works on three levels of package management structure: package providers, package sources, and the packages themselves.

Term	Description
Package manager	Software package management system. In PackageManagement terms, this is a package provider.
Package provider	PackageManagement term for a package manager. Examples can include Windows Installer, Chocolatey, and others.
Package source	A URL, local folder, or network shared folder that you configure package providers to use as a repository.
Package	A piece of software that a package provider manages, and that is stored in a specific package source.

The PackageManagement module includes the following cmdlets. You can find the Help for these cmdlets on TechNet starting on the following page: http://technet.microsoft.com/library/dn890951(v=wps.640).aspx.

Cmdlet	Description
Get-PackageProvider	Returns a list of package providers that are connected to PackageManagement.
Get-PackageSource	Gets a list of package sources that are registered for a package provider.
Register-PackageSource	Adds a package source for a specified

package provider.
Set-PackageSource Sets properties on an existing package source.
Unregister-PackageSource Removes a registered package source.
Get-Package Returns a list of installed software packages.
Find-Package Finds software packages in available package sources.
Install-Package Installs one or more software packages.
Save-Package Saves packages to the local computer without installing them.
Uninstall-Package Uninstalls one or more software packages.

PackageManagement Package Provider Bootstrapping and Dynamic Cmdlet Parameters

By default, PackageManagement ships with a core bootstrap provider. You can install additional package providers as you need them by bootstrapping the providers; that is, responding to a prompt to install the provider automatically, from a web service. You can specify a package provider with any PackageManagement cmdlet; if the specified provider is not available, PackageManagement prompts you to bootstrap --or automatically install--the provider. In the following examples, if the Chocolatey provider is not already installed, PackageManagement bootstrapping installs the provider.

```
Find-Package –Provider Chocolatey <PackageName>
```

If the Chocolatey provider is not already installed, when you run the preceding command, you are prompted to install it.

```
Install-Package <Chocolatey package Name> -ForceBootstrap
```

If the Chocolatey provider is not already installed, when you run the preceding command, the provider is installed; but because the ForceBootstrap parameter has been added to the command, you are not prompted to install it; both the provider and the package are installed automatically.

When you try to install a package, if you do not already have the supporting provider installed, and you do not add the ForceBootstrap parameter to your command, PackageManagement prompts you to install the provider.

Specifying a package provider in your PackageManagement command can make dynamic parameters available that are specific to that package provider. When you run Get-Help for a specific PackageManagement cmdlet, a list of parameter sets are returned, grouping dynamic parameters for available package providers in separate parameter sets.

More Information About the PackageManagement Project

For more information about the PackageManagement open development project, including how to create a PackageManagement package provider, see the PackageManagement project on GitHub at https://oneget.org.

SEE ALSO

Get-PackageProvider
Get-PackageSource
Register-PackageSource
Set-PackageSource
Unregister-PackageSource
Get-Package
Find-Package
Install-Package
Save-Package
Uninstall-Package

Name	Category	Module	Synopsis
about_Parallel	HelpFile		Describes the Parallel keyword, which runs the
about_Parallel	HelpFile		Describes the Parallel keyword, which runs the

TOPIC

about_Parameters

SHORT DESCRIPTION

Describes how to work with command parameters in Windows PowerShell.

LONG DESCRIPTION

Most Windows PowerShell commands, such as cmdlets, functions, and scripts, rely on parameters to allow users to select options or provide input. The parameters follow the command name and have the following form:

-<parameter_name> <parameter_value>

The name of the parameter is preceded by a hyphen (-), which signals to Windows PowerShell that the word following the hyphen is a parameter name. Some parameters do not require or accept a parameter value. Other parameters require a value, but do not require the parameter name in the command.

The type of parameters and the requirements for those parameters vary. To find information about the parameters of a command, use the Get-Help cmdlet. For example, to find information about the parameters of the Get-ChildItem cmdlet, type:

Get-Help Get-ChildItem

To find information about the parameters of a script, use the full path to the script file. For example:

Get-Help $home\Documents\Scripts\Get-Function.ps1

The Get-Help cmdlet returns various details about the command, including a description, the command syntax, information about the parameters, and examples showing how to use the parameters in a command.

You can also use the Parameter parameter of the Get-Help cmdlet to find information about a particular parameter. Or, you can use the Parameter parameter with the wildcard character (*) value to find information about all parameters of the command. For example, the following command gets information about all parameters of the Get-Member cmdlet:

Get-Help Get-Member -Parameter *

DEFAULT PARAMETER VALUES

Optional parameters have a default value, which is the value that is used or assumed when the parameter is not specified in the command.

For example, the default value of the ComputerName parameter of many cmdlets is the name of the local computer. As a result, the local computer name is used in the command unless the ComputerName parameter is specified.

To find the default parameter value, see help topic for the cmdlet. The parameter description should include the default value.

You can also set a custom default value for any parameter of a cmdlet or advanced function. For information about setting custom default values, see about_Parameters_Default_Values.

PARAMETER ATTRIBUTE TABLE

When you use the Full, Parameter, or Online parameters of the Get-Help cmdlet, Get-Help displays a parameter attribute table with detailed information about the parameter.

This information includes the details you need to know to use the parameter. For example, the help topic for the Get-ChildItem cmdlet includes the following details about its Path parameter:

```
-path <string[]>
    Specifies a path of one or more locations. Wildcard characters are
    permitted. The default location is the current directory (.).

Required?                    false
Position?                    1
Default value                Current directory
Accept pipeline input?       true (ByValue, ByPropertyName)
Accept wildcard characters?  true
```

The parameter information includes the parameter syntax, a description of the parameter, and the parameter attributes. The following sections describe the parameter attributes.

Parameter Required?

This setting indicates whether the parameter is mandatory, that is, whether all commands that use this cmdlet must include this parameter. When the value is "True" and the parameter is missing from the command, Windows PowerShell prompts you for a value for the parameter.

Parameter Position?

This setting indicates whether you can supply a parameter's value without preceding it with the parameter name. If set to "0" or "named," a parameter name is required. This type of parameter is referred to as a named parameter. A named parameter can be listed in any position after the cmdlet name.

If the "Parameter position?" setting is set to an integer other than 0, the parameter name is not required. This type of parameter is referred to as a positional parameter, and the number indicates the position in which the parameter must appear in relation to other positional parameters. If you include the parameter name for a positional parameter, the parameter can be listed in any position after the cmdlet name.

For example, the Get-ChildItem cmdlet has Path and Exclude parameters. The "Parameter position?" setting for Path is 1, which means that it is a positional parameter. The "Parameter position?" setting for Exclude is 0, which means that it is a named parameter.

This means that Path does not require the parameter name, but its parameter value must be the first or only unnamed parameter value in the command. However, because the Exclude parameter is a named parameter, you can place it in any position in the command.

As a result of the "Parameter position?" settings for these two parameters, you can use any of the following commands:

```
Get-ChildItem -path c:\techdocs -exclude *.ppt
Get-ChildItem c:\techdocs -exclude *.ppt
Get-ChildItem -exclude *.ppt -path c:\techdocs
Get-ChildItem -exclude *.ppt c:\techdocs
```

If you were to include another positional parameter without including the parameter name, that parameter would have to be placed in the order specified by the "Parameter position?" setting.

Parameter Type

This setting specifies the Microsoft .NET Framework type of the parameter value. For example, if the type is Int32, the parameter value must be an integer. If the type is string, the parameter value must be a character string. If the string contains spaces, the value must be enclosed in quotation marks, or the spaces must be preceded by the escape character (`).

Default Value

This setting specifies the value that the parameter will assume if no other value is provided. For example, the default value of

the Path parameter is often the current directory. Required parameters never have a default value. For many optional parameters, there is no default because the parameter has no effect if it is not used.

Accepts Multiple Values?

This setting indicates whether a parameter accepts multiple parameter values. When a parameter accepts multiple values, you can type a comma-separated list as the value of the parameter in the command, or save a comma-separated list (an array) in a variable, and then specify the variable as the parameter value.

For example, the ServiceName parameter of the Get-Service cmdlet accepts multiple values. The following commands are both valid:

```
get-service -servicename winrm, netlogon
```

```
$s = "winrm", "netlogon"
get-service -servicename $s
```

Accepts Pipeline Input?

This setting indicates whether you can use the pipeline operator (|) to send a value to the parameter.

Value	Description
False	Indicates that you cannot pipe a value to the parameter.
True (by Value)	Indicates that you can pipe any value to the parameter, just so the value has the .NET Framework type specified for the parameter or the value can be converted to the specified .NET Framework type. When a parameter is "True (by Value)", Windows PowerShell tries to associate any piped values with that parameter before it tries other methods to interpret the command.

True (by Property Name) Indicates that you can pipe a value to the
parameter, but the .NET Framework type of the
parameter must include a property with the same
name as the parameter.

For example, you can pipe a value to a Name
parameter only when the value has a property
called "Name".

Accepts Wildcard Characters?
This setting indicates whether the parameter's value can contain
wildcard characters so that the parameter value can be matched to more
than one existing item in the target container.

Common Parameters
Common parameters are parameters that you can use with any cmdlet.
For more information about common parameters, see about_CommonParameters

SEE ALSO
about_Command_syntax
about_Comment_Based_Help
about_Functions_Advanced
about_Parameters_Default_Values
about_Pipelines
about_Wildcards

TOPIC

about_Parameters_Default_Values

SHORT DESCRIPTION
Describes how to set custom default values for the parameters of cmdlets and
advanced functions.

LONG DESCRIPTION
The $PSDefaultParameterValues preference variable lets you specify custom
default values for any cmdlet or advanced function. Cmdlets and functions
use the custom default value unless you specify another value in the command.

The authors of cmdlets and advanced functions set standard default values for
their parameters. Typically, the standard default values are useful, but they
might not be appropriate for all environments.

This feature is especially useful when you must specify the same alternate parameter value nearly every time you use the command or when a particular parameter value is difficult to remember, such as an e-mail server name or project GUID.

If the desired default value varies predictably, you can specify a script block that provides different default values for a parameter under different conditions.

$PSDefaultParameterValues was introduced in Windows PowerShell 3.0.

SYNTAX

The syntax of the $PSDefaultParameterValues preference variable is as follows:

```
$PSDefaultParameterValues=@{"<CmdletName>:<ParameterName>"="<DefaultValue>"}

$PSDefaultParameterValues=@{"<CmdletName>:<ParameterName>"={<ScriptBlock>}}

$PSDefaultParameterValues["Disabled"]=$true | $false
```

Wildcard characters are permitted in the CmdletName and ParameterName values.

The value of $PSDefaultParameterValues is a **System.Management.Automation.DefaultParameterDictionary,** a type of hash table that validates the format of keys. Enter a hash table where the key consists of the cmdlet name and parameter name separated by a colon (:) and the value is the custom default value.

To set, change, add, or remove entries from $PSDefaultParameterValues, use the methods that you would use to edit a standard hash table.

The <CmdletName> must be the name of a cmdlet or the name of an advanced function that uses the CmdletBinding attribute. You cannot use $PSDefaultParameterValues to specify default values for scripts or simple functions.

The default value can be an object or a script block. If the value is a script block, Windows PowerShell evaluates the script block and uses the result as the parameter value. When the specified parameter takes a script block value, enclose the script block value in a second set of braces, such as:
$PSDefaultParameterValues=@{ "Invoke-Command:ScriptBlock"={{Get-Process}} }

For information about hash tables, see about_Hash_Tables. For information about script blocks, see about_Script_Blocks. For information about preference variables, see about_Preference_Variables.

EXAMPLES

The following command sets a custom default value for the SmtpServer parameter of the Send-MailMessage cmdlet.

```
$PSDefaultParameterValues = @{"Send-MailMessage:SmtpServer"="Server01AB234x5"}
```

To set default values for multiple parameters, use a semi-colon (;) to separate each Name=Value pair. The following command adds a custom default value for the LogName parameter of the Get-WinEvent cmdlet.

```
$PSDefaultParameterValues = @{"Send-MailMessage:SmtpServer"="Server01AB234x5";
    "Get-WinEvent:LogName"="Microsoft-Windows-PrintService/Operational"}
```

You can use wildcard characters in the name of the cmdlet and parameter. The following command sets the Verbose common parameter to $true in all commands. Use $true and $false to set values for switch parameters, such as Verbose.

```
$PSDefaultParameterValues = @{"*:Verbose"=$true}
```

If a parameter takes multiple values (an array), you can set multiple values as the default value. The following command sets the default value of the ComputerName parameter of the Invoke-Command cmdlet to "Server01" and "Server02".

```
$PSDefaultParameterValues = @{"Invoke-Command:ComputerName"="Server01","Server02"}
```

You can use a script block to specify different default values for a parameter under different conditions. Windows PowerShell evaluates the script block and uses the result as the default parameter value.

The following command sets the default value of the Autosize parameter of the Format-Table cmdlet to $true when the host program is the Windows PowerShell console.

```
$PSDefaultParameterValues=@{"Format-Table:AutoSize"={if ($host.Name –eq
"ConsoleHost"){$true}}}
```

If a parameter takes a script block value, enclose the script block in an extra set of braces. When Windows PowerShell evaluates the outer script block, the result is the inner script block, which is set as the default parameter value.

The following command sets the default value of the ScriptBlock parameter of Invoke-Command. Because the script block is enclosed in a second set of braces, the enclosed script block is passed to the Invoke-Command cmdlet.

```
$PSDefaultParameterValues=@{"Invoke-Command:ScriptBlock"={{Get-EventLog –Log System}}}
```

HOW TO SET $PSDefaultParameterValues

$PSDefaultParameterValues is a preference variable, so it exists only in the session in which it is set. It has no default value.

To set $PSDefaultParameterValues, type the variable name and one or more key-value pairs at the command line.

If you type another $PSDefaultParameterValues command, its value replaces the original value. The original is not retained.

To save $PSDefaultParameterValues for future sessions, add a $PSDefaultParameterValues command to your Windows PowerShell profile. For more information, see about_Profiles.

HOW TO GET $PSDefaultParameterValues

To get the value of $PSDefaultParameterValues, at the command prompt, type:
$PSDefaultParameterValues

For example, the first command sets the value of $PSDefaultParameterValues. The second command gets the value of $PSDefaultParameterValues.

```
PS C:\> $PSDefaultParameterValues = @{"Send-MailMessage:SmtpServer"="Server01AB234x5";
      "Get-WinEvent:LogName"="Microsoft-Windows-PrintService/Operational";
      "Get-*:Verbose"=$true}

PS C:\> $PSDefaultParameterValues

Name                           Value
----                           -----
Send-MailMessage:SmtpServer    Server01AB234x5
Get-WinEvent:LogName           Microsoft-Windows-PrintService/Operational
Get*:Verbose                   True
```

To get the value of a <CmdletName:ParameterName> key, use the following command syntax:

```
$PSDefaultParameterValues["<CmdletName:ParameterName>"]
```

For example:

```
PS C:\> $PSDefaultParameterValues["Send-MailMessage:SmtpServer"]
```

Server01AB234x5

HOW TO ADD VALUES TO $PSDefaultParameterValues

To add or remove values from $PSDefaultParameterValues, use the methods that you would use for a standard hash table.

For example, to add a value to $PSDefaultParameterValues without affecting the existing values, use the Add method of hash tables.

The syntax of the Add method is as follows:

```
<HashTable>.Add(Key, Value)
```

where the Key is "<CmdletName>:<ParameterName>" and the value is the parameter value.

Use the following command format to call the Add method on $PSDefaultParameterValues. Be sure to use a comma (,) to separate the key from the value, instead of the equal sign (=).

```
$PSDefaultParameterValues.Add("<CmdletName>:<ParameterName>", "<ParameterValue>")
```

For example, the following command adds "PowerShell" as the default value of the Name parameter of the Get-Process cmdlet.

```
$PSDefaultParameterValues.Add("Get-Process:Name", "PowerShell")
```

The following example shows the effect of this command.

```
PS C:\> $PSDefaultParameterValues

Name                           Value
----                           -----
Send-MailMessage:SmtpServer    Server01AB234x5
Get-WinEvent:LogName           Microsoft-Windows-PrintService/Operational
Get*:Verbose                   True

PS C:\> $PSDefaultParameterValues.Add("Get-Process:Name", "PowerShell")

PS C:\> $PSDefaultParameterValues

Name                           Value
----                           -----
Get-Process:Name               PowerShell
Send-MailMessage:SmtpServer    Server01AB234x5
```

```
Get-WinEvent:LogName          Microsoft-Windows-PrintService/Operational
Get*:Verbose                  True
```

HOW TO REMOVE VALUES FROM $PSDefaultParameterValues

To remove a value from $PSDefaultParameterValues, use the Remove method of hash tables.

The syntax of the Remove method is as follows:

```
<HashTable>.Remove(Key)
```

Use the following command format to call the Remove method on $PSDefaultParameterValues.

```
$PSDefaultParameterValues.Remove("<CmdletName>:<ParameterName>")
```

For example, the following command removes the Name parameter of the Get-Process cmdlet and its values.

```
$PSDefaultParameterValues.Remove("Get-Process:Name")
```

The following example shows the effect of this command.

```
PS C:\> $PSDefaultParameterValues

Name                      Value
----                      -----
Get-Process:Name          PowerShell
Send-MailMessage:SmtpServer    Server01AB234x5
Get-WinEvent:LogName          Microsoft-Windows-PrintService/Operational
Get*:Verbose                  True

PS C:\> $PSDefaultParameterValues.Remove("Get-Process:Name")

PS C:\> $PSDefaultParameterValues

Name                      Value
----                      -----
Send-MailMessage:SmtpServer    Server01AB234x5
Get-WinEvent:LogName          Microsoft-Windows-PrintService/Operational
Get*:Verbose                  True
```

HOW TO CHANGE VALUES IN $PSDefaultParameterValues

To change a value in $PSDefaultParameterValues, use the following command format.

```
$PSDefaultParameterValues["CmdletName:ParameterName"]="<NewValue>"
```

The following example shows the effect of this command.

```
PS C:\> $PSDefaultParameterValues

Name                           Value
----                           -----
Send-MailMessage:SmtpServer    Server01AB234x5
Get-WinEvent:LogName           Microsoft-Windows-PrintService/Operational
Get*:Verbose                   True

PS C:\> $PSDefaultParameterValues["Send-MailMessage:SmtpServer"]="Server0000cabx5"

PS C:\> $PSDefaultParameterValues

Name                           Value
----                           -----
Send-MailMessage:SmtpServer    Server0000cabx5
Get-WinEvent:LogName           Microsoft-Windows-PrintService/Operational
Get*:Verbose                   True
```

HOW TO DISABLE AND RE-ENABLE $PSDefaultParameterValues

You can temporarily disable and then re-enable $PSDefaultParameterValues. This is very useful if you're running scripts that might assume different default parameter values.

To disable $PSDefaultParameterValues, add a key of "Disabled" with a value of $True.

For example,

```
$PSDefaultParameterValues.Add("Disabled", $true)
```

- or -

```
$PSDefaultParameterValues[Disabled]=$true
```

The other values in $PSDefaultParameterValues are preserved, but not effective.

```
PS C:\> $PSDefaultParameterValues
```

```
    Name                           Value
    ----                           -----
    Disabled                       True
    Send-MailMessage:SmtpServer    Server0000cabx5
    Get-WinEvent:LogName           Microsoft-Windows-PrintService/Operational
    Get*:Verbose                   True
```

To re-enable $PSDefaultParameterValues, remove the Disabled key or change the value of the Disabled key to $False.

```
    $PSDefaultParameterValues.Remove("Disabled")
```

- or -

```
    $PSDefaultParameterValues[Disabled]=$false
```

The previous value of $PSDefaultParameterValues is effective again.

KEYWORDS

about_PSDefaultParameterValues
about_Parameters_DefaultValues
about_DefaultValues

SEE ALSO

about_Hash_Tables
about_Preference_Variables
about_Profiles
about_Script_Blocks

TOPIC

about_Parsing

SHORT DESCRIPTION

Describes how Windows PowerShell parses commands.

LONG DESCRIPTION

When you enter a command at the command prompt, Windows PowerShell breaks the command text into a series of segments called "tokens" and then determines how to interpret each "token."

For example, if you type:

```
Write-Host book
```

Windows PowerShell breaks the following command into two tokens, "Write-Host" and "book", and interprets each token independently.

When processing a command, the Windows PowerShell parser operates in expression mode or in argument mode:

- In expression mode, character string values must be contained in quotation marks. Numbers not enclosed in quotation marks are treated as numerical values (rather than as a series of characters).

- In argument mode, each value is treated as an expandable string unless it begins with one of the following special characters: dollar sign ($), at sign (@), single quotation mark ('), double quotation mark ("), or an opening parenthesis (().

If preceded by one of these characters, the value is treated as a value expression.

The following table provides several examples of commands processed in expression mode and argument mode and the results produced by those commands.

Example	Mode	Result
2+2	Expression	4 (integer)
Write-Output 2+2	Argument	"2+2" (string)
Write-Output (2+2)	Expression	4 (integer)
$a = 2+2	Expression	$a = 4 (integer)
Write-Output $a	Expression	4 (integer)
Write-Output $a/H	Argument	"4/H" (string)

Every token can be interpreted as some kind of object type, such as Boolean or string. Windows PowerShell attempts to determine the

object type from the expression. The object type depends on the type of parameter a command expects and on whether Windows PowerShell knows how to convert the argument to the correct type. The following table shows several examples of the types assigned to values returned by the expressions.

Example	Mode	Result
Write-Output !1	argument	"!1" (string)
Write-Output (!1)	expression	False (Boolean)
Write-Output (2)	expression	2 (integer)

STOP PARSING: --%

The stop-parsing symbol (--%), introduced in Windows PowerShell 3.0, directs Windows PowerShell to refrain from interpreting input as Windows PowerShell commands or expressions.

When calling an executable program in Windows PowerShell, place the stop-parsing symbol before the program arguments. This technique is much easier than using escape characters to prevent misinterpretation.

When it encounters a stop-parsing symbol, Windows PowerShell treats the remaining characters in the line as a literal. The only interpretation it performs is to substitute values for environment variables that use standard Windows notation, such as %USERPROFILE%.

The stop-parsing symbol is effective only until the next newline or pipeline character. You cannot use a continuation character (`) to extend its effect or use a command delimiter (;) to terminate its effect.

For example, the following command calls the Icacls program.

```
icacls X:\VMS /grant Dom\HVAdmin:(CI)(OI)F
```

To run this command in Windows PowerShell 2.0, you must use escape characters to prevent Windows PowerShell from misinterpreting the parentheses.

```
icacls X:\VMS /grant Dom\HVAdmin:`(CI`)`(OI`)F
```

Beginning in Windows PowerShell 3.0, you can use the stop-parsing symbol.

```
icacls X:\VMS --% /grant Dom\HVAdmin:(CI)(OI)F
```

Windows PowerShell sends the following command string to the Icacls program:

```
X:\VMS /grant Dom\HVAdmin:(CI)(OI)F
```

SEE ALSO
about_Command_Syntax

TOPIC

about_Parsing

SHORT DESCRIPTION
Describes how Windows PowerShell parses commands.

LONG DESCRIPTION
When you enter a command at the command prompt, Windows PowerShell breaks the command text into a series of segments called tokens, and then determines how to interpret each token.

For example, if you type:

Write-Host book

Windows PowerShell breaks the following command into two tokens, "Write-Host" and "book", and interprets each token independently.

When processing a command, the Windows PowerShell parser operates in expression mode or in argument mode:

In expression mode, character string values must be contained in quotation marks. Numbers not enclosed in quotation marks are treated as numerical values (rather than as a series of characters).
In argument mode, each value is treated as an expandable string unless it begins with one of the following special characters: dollar sign ($), at sign (@), single quotation mark ('), double quotation mark ("), or an opening parenthesis (().

If preceded by one of these characters, the value is treated as a value expression.

The following table provides several examples of commands processed in expression mode and argument mode and the results produced by those commands.

```
Example          Mode        Result
---------------- ----------  ----------------
2+2              Expression  4 (integer)
Write-Output 2+2 Argument    "2+2" (string)
Write-Output (2+2) Expression  4 (integer)
$a = 2+2         Expression  $a = 4 (integer)
Write-Output $a  Expression  4 (integer)
Write-Output $a/H  Argument  "4/H" (string)
```

Every token can be interpreted as some kind of object type, such as Boolean or string. Windows PowerShell attempts to determine the object type from the expression. The object type depends on the type of parameter a command expects and on whether Windows PowerShell knows how to convert the argument to the correct type. The following table shows several examples of the types assigned to values returned by the expressions.

```
Example          Mode        Result
---------------- ----------  ---------------
Write-Output !1  argument    "!1" (string)
Write-Output (!1)  expression  False (Boolean)
Write-Output (2)  expression  2 (integer)
```

STOP PARSING: --%

The stop-parsing symbol (--%), introduced in Windows PowerShell 3.0, directs Windows PowerShell to refrain from interpreting input as Windows PowerShell commands or expressions.

When calling an executable program in Windows PowerShell, place the stop-parsing symbol before the program arguments. This technique is much easier than using escape characters to prevent misinterpretation.

When it encounters a stop-parsing symbol, Windows PowerShell treats the remaining characters in the line as a literal. The only interpretation it performs is to substitute values for environment variables that use standard Windows notation, such as %USERPROFILE%.

The stop-parsing symbol is effective only until the next newline or pipeline character. You cannot use a continuation character (`) to extend its effect or use a command delimiter (;) to terminate its effect.

For example, the following command calls the Icacls program.

```
icacls X:\VMS /grant Dom\HVAdmin:(CI)(OI)F
```

To run this command in Windows PowerShell 2.0, you must use escape characters to prevent Windows PowerShell from misinterpreting the parentheses, as shown in the following example.

```
icacls X:\VMS /grant Dom\HVAdmin:`(CI`)`(OI`)F
```

Beginning in Windows PowerShell 3.0, you can use the stop-parsing symbol, as shown in the following example.

icacls X:\VMS --% /grant Dom\HVAdmin:(CI)(OI)F

Windows PowerShell sends the following command string to the Icacls program:

X:\VMS /grant Dom\HVAdmin:(CI)(OI)F

SEE ALSO
about_Command_Syntax

TOPIC

about_Path_Syntax

SHORT DESCRIPTION
Describes the full and relative path name formats in Windows PowerShell.

LONG DESCRIPTION
All items in a data store accessible through a Windows PowerShell provider can be uniquely identified 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 Windows PowerShell drive through which the containers are accessed.

In Windows PowerShell, 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:

[<provider>::]<drive>:[\<container>[\<subcontainer>...]]\<item>

The <provider> placeholder refers to the Windows PowerShell provider through which you access the data store. For example, the FileSystem

provider allows you to access the files and directories on your computer. This element of the syntax is optional and is never needed because the drive names are unique across all providers.

The <drive> placeholder refers to the Windows PowerShell drive that is supported by a particular Windows PowerShell provider. In the case of the FileSystem provider, the Windows PowerShell drives map to the Windows drives that are configured on your system. For example, if your system includes an A: drive and a C: drive, the FileSystem provider creates the same drives in Windows PowerShell.

After you have specified the drive, you must specify any containers and subcontainers that contain the item. The containers must be specified in the hierarchical order in which they exist in the data store. In other words, you must start with the parent container, then the child container in that parent container, and so on. In addition, each container must be preceded by a backslash. (Note that Windows PowerShell allows you to use forward slashes for compatibility with other PowerShells.)

After the container and subcontainers have been specified, you must provide the item name, preceded by a backslash. For example, the fully qualified path name for the Shell.dll file in the C:\Windows\System32 directory is as follows:

```
C:\Windows\System32\Shell.dll
```

In this case, the drive through which the containers are accessed is the C: drive, the top-level container is Windows, the subcontainer is System32 (located within the Windows container), and the item is Shell.dll.

In some situations, you do not need to specify a fully qualified path name and can instead use a relative path name. A relative path name is based on the current working location. Windows PowerShell allows you to identify an item based on its location relative to the current working location. You can specify relative path names by using 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.

Symbol	Description	Relative path	Fully qualified path
.	Current working location	.\System	c:\Windows\System
..	Parent of current working location	..\Program Files	c:\Program Files
\	Drive root of current working location	\Program Files	c:\Program Files
[none]	No special characters	System	c:\Windows\System

When using a path name in a command, you enter that name in the same way whether you use a fully qualified path name or a relative one. For example, suppose that your current working directory is C:\Windows. The following Get-ChildItem command retrieves all items in the C:\Techdocs directory:

```
Get-ChildItem \techdocs
```

The backslash indicates that the drive root of the current working location should be used. Because the working directory is C:\Windows, the drive root is the C: drive. Because the techdocs directory is located off the root, you need to specify only the backslash.

You can achieve the same results by using the following command:

```
Get-ChildItem c:\techdocs
```

Regardless of whether you use a fully qualified path name or a relative path name, a path name is important not only because it locates an item but also because it uniquely identifies the item even if that item shares the same name as another item in a different container.

For instance, suppose that you have two files that are each named Results.txt. The first file is in a directory named C:\Techdocs\Jan, and the second file is in a directory named C:\Techdocs\Feb. The path name for the first file (C:\Techdocs\Jan\Results.txt) and the path name for the second file (C:\Techdocs\Feb\Results.txt) allow you to clearly distinguish between the two files.

SEE ALSO

about_Locations

TOPIC

about_pipelines

SHORT DESCRIPTION

Combining commands into pipelines in the Windows PowerShell

LONG DESCRIPTION

A pipeline is a series of commands connected by pipeline operators (|)(ASCII 124). Each pipeline operator sends the results of the preceding command to the next command.

You can use pipelines to send the objects that are output by one command to be used as input to another command for processing. And you can send the output of that command to yet another command. The result is a very powerful command chain or "pipeline" that is comprised of a series of simple commands.

For example,

```
Command-1 | Command-2 | Command-3
```

In this example, the objects that Command-1 emits are sent to Command-2. Command-2 processes the objects and sends them to Command-3. Command-3 processes the objects and send them down the pipeline. Because there are no more commands in the pipeline, the results are displayed at the console.

In a pipeline, the commands are processed from left to right in the order that they appear. The processing is handled as a single operation and output is displayed as it is generated.

Here is a simple example. The following command gets the Notepad process and then stops it.

```
get-process notepad | stop-process
```

The first command uses the Get-Process cmdlet to get an object representing the Notepad process. It uses a pipeline operator (|) to send the process object to the Stop-Process cmdlet, which stops the Notepad process. Notice that the Stop-Process command does not have a Name or ID parameter to specify the process, because the specified process is submitted through the pipeline.

Here is a practical example. This command pipeline gets the text files in the current directory, selects only the files that are more than 10,000 bytes long, sorts them by length, and displays the name and length of each file in a table.

```
Get-ChildItem -path *.txt | Where-Object {$_.length -gt 10000} |
Sort-Object -property Length | Format-Table -property name, length
```

This pipeline is comprised of four commands in the specified order. The command is written horizontally, but we will show the process vertically in the following graphic.

```
Get-ChildItem -path *.txt

        |
        |  (FileInfo objects )
        |  (    .txt        )
        |
        V

Where-Object {$_.length -gt 10000}

        |
        |  (FileInfo objects )
        |  (    .txt        )
        |  ( Length > 10000  )
        |
        V

Sort-Object -property Length

        |
        |  (FileInfo objects  )
        |  (    .txt        )
        |  ( Length > 10000   )
        |  ( Sorted by length )
        |
        V

Format-Table -property name, length
```

```
        |
        |  (FileInfo objects     )
        |  (    .txt           )
        |  ( Length > 10000      )
        |  ( Sorted by length    )
        |  (Formatted in a table )
        |
        V
    Name                Length
    ----                ------
    tmp1.txt             82920
    tmp2.txt            114000
    tmp3.txt            114000
```

USING PIPELINES

The Windows PowerShell cmdlets were designed to be used in pipelines. For example, you can usually pipe the results of a Get cmdlet to an action cmdlet (such as a Set, Start, Stop, or Rename cmdlet) for the same noun.

For example, you can pipe any service from the Get-Service cmdlet to the Start-Service or Stop-Service cmdlets (although disabled services cannot be restarted in this way).

This command pipeline starts the WMI service on the computer:

```
get-service wmi | start-service
```

The cmdlets that get and set objects of the Windows PowerShell providers, such as the Item and ItemProperty cmdlets, are also designed to be used in pipelines.

For example, you can pipe the results of a Get-Item or Get-ChildItem command in the Windows PowerShell registry provider to the New-ItemProperty cmdlet. This command adds a new registry entry, NoOfEmployees, with a value of 8124, to the MyCompany registry key.

```
  get-item -path HKLM:\Software\MyCompany | new-Itemproperty -name NoOfEmployees -value
8124
```

Many of the utility cmdlets, such as Get-Member, Where-Object, Sort-Object, Group-Object, and Measure-Object are used almost exclusively in pipelines. You can pipe any objects to these cmdlets.

For example, you can pipe all of the processes on the computer to the Sort-Object command and have them sorted by the number of handles in the process.

```
get-process | sort-object -property handles
```

Also, you can pipe any objects to the formatting cmdlets, such as Format-List and

Format-Table, the Export cmdlets, such as Export-Clixml and Export-CSV, and the Out cmdlets, such as Out-Printer.

For example, you can pipe the Winlogon process to the Format-List cmdlet to display all of the properties of the process in a list.

```
get-process winlogon | format-list -property *
```

With a bit of practice, you'll find that combining simple commands into pipelines saves time and typing, and makes your scripting more efficient.

HOW PIPELINES WORK

When you "pipe" objects, that is send the objects in the output of one command to another command, Windows PowerShell tries to associate the piped objects with one of the parameters of the receiving cmdlet.

To do so, the Windows PowerShell "parameter binding" component, which associates input objects with cmdlet parameters, tries to find a parameter that meets the following criteria:

-- The parameter must accept input from a pipeline (not all do)
-- The parameter must accept the type of object being sent or a type that the object can be converted to.
-- The parameter must not already be used in the command.

For example, the Start-Service cmdlet has many parameters, but only two of them, Name and InputObject accept pipeline input. The Name parameter takes strings and the InputObject parameter takes service objects. Therefore, you can pipe strings and service objects (and objects with properties that can be converted to string and service objects) to Start-Service.

If the parameter binding component of Windows PowerShell cannot associate the piped objects with a parameter of the receiving cmdlet, the command fails and Windows PowerShell prompts you for the missing parameter values.

You cannot force the parameter binding component to associate the piped objects with a particular parameter -- you cannot even suggest a parameter. Instead, the logic of the component manages the piping as efficiently as possible.

ONE-AT-A-TIME PROCESSING

Piping objects to a command is much like using a parameter of the command to submit the objects.

For example, piping objects representing the services on the computer to a Format-Table command, such as:

```
get-service | format-table -property name, dependentservices
```

is much like saving the service objects in a variable and using the InputObject parameter of Format-Table to submit the service object.

```
$services = get-service
        format-table -inputobject $services -property name, dependentservices
```

or imbedding the command in the parameter value

```
        format-table -inputobject (get-service wmi) -property name, dependentservices
```

However, there is an important difference. When you pipe multiple objects to a command, Windows PowerShell sends the objects to the command one at a time. When you use a command parameter, the objects are sent as a single array object.

This seemingly technical difference can have interesting, and sometimes useful, consequences.

For example, if you pipe multiple process objects from the Get-Process cmdlet to the Get-Member cmdlet, Windows PowerShell sends each process object, one at a time, to Get-Member. Get-Member displays the .NET class (type) of the process objects, and their properties and methods. (Get-Member eliminates duplicates, so if the objects are all of the same type, it displays only one object type.)

In this case, Get-Member displays the properties and methods of each process object, that is, a System.Diagnostics.Process object.

```
get-process | get-member

  TypeName: System.Diagnostics.Process

Name                   MemberType     Definition
----                   ----------     ----------
Handles                AliasProperty  Handles = Handlecount
Name                   AliasProperty  Name = ProcessName
NPM                    AliasProperty  NPM = NonpagedSystemMemorySize
...
```

However, if you use the InputObject parameter of Get-Member, then Get-Member receives an array of System.Diagnostics.Process objects as a single unit, and it displays the properties of an array of objects. (Note the array symbol ([]) after the System.Object type name.)

```
get-member -inputobject (get-process)
```

TypeName: System.Object[]

```
Name            MemberType    Definition
----            ----------    ----------
Count           AliasProperty Count = Length
Address         Method        System.Object& Address(Int32 )
Clone           Method        System.Object Clone()
...
```

This result might not be what you intended, but after you understand it, you can use it. For example, an array of process objects has a Count property that you can use to count the number of processes on the computer.

```
(get-process).count
```

This distinction can be important, so remember that when you pipe objects to a cmdlet, they are delivered one at a time.

ACCEPTS PIPELINE INPUT

In order to receive objects in a pipeline, the receiving cmdlet must have a parameter that accepts pipeline input. You can use a Get-Help command with the Full or Parameter parameters to determine which, if any, of a cmdlet's parameters accepts pipeline input.

In the Get-Help default display, the "Accepts pipeline input" item appears in a table of parameter attributes. This table is displayed only when you use the Full or Parameter parameters of the Get-Help cmdlet.

For example, to determine which of the parameters of the Start-Service cmdlet accepts pipeline input, type:

```
get-help start-service -full

get-help start-service -parameter *
```

For example, the help for the Start-Service cmdlet shows that the Name and InputObject parameters accept pipeline input ("true"). All other parameters have a value of "false" in the "Accept pipeline input?" row.

```
-name <string[]>
  Specifies the service names for the service to be started.
  The parameter name is optional. You can use "-Name" or its alias,
  "-ServiceName", or you can omit the parameter name.

  Required?               true
  Position?               1
  Default value
```

--> Accept pipeline input? true (ByValue, ByPropertyName)
Accept wildcard characters? true

-inputObject <ServiceController[]>
Specifies ServiceController objects representing the services to be started. Enter a variable that contains the objects or type a command or expression that gets the objects.

Required? false
Position? named
Default value
--> Accept pipeline input? true (ByValue)
Accept wildcard characters? false

This means that you can send objects (PsObjects) through the pipeline to the Where-Object cmdlet and Windows PowerShell will associate the object with the InputObject parameter.

METHODS OF ACCEPTING PIPELINE INPUT

Cmdlets parameters can accept pipeline input in one of two different ways:

-- ByValue: Parameters that accept input "by value" can accept piped objects that have the same .NET type as their parameter value or objects that can be converted to that type.

For example, the Name parameter of Start-Service accepts pipeline input by value. It can accept string objects or objects that can be converted to strings.

-- ByPropertyName: Parameters that accept input "by property name" can accept piped objects only when a property of the object has the same name as the parameter.

For example, the Name parameter of Start-Service can accept objects that have a Name property.

(To list the properties of an object, pipe it to Get-Member.)

Some parameters can accept objects by value or by property name. These parameters are designed to take input from the pipeline easily.

INVESTIGATING PIPELINE ERRORS

If a command fails because of a pipeline error, you can investigate the failure and rewrite the command.

For example, the following command tries to move a registry entry from one registry key to another by using the Get-Item cmdlet to get the destination path and then piping the path to the Move-ItemProperty cmdlet.

Specifically, the command uses the Get-Item cmdlet to get the destination path. It uses a pipeline operator to send the result to the Move-ItemProperty cmdlet. The Move-ItemProperty command specifies the current path and name of the registry entry to be moved.

```
get-item -path hklm:\software\mycompany\sales |
move-itemproperty -path hklm:\software\mycompany\design -name product
```

The command fails and Windows PowerShell displays the following error message:

```
Move-ItemProperty : The input object cannot be bound to any parameters for the
command either because the command does not take pipeline input or the input
and its properties do not match any of the parameters that take pipeline input.
At line:1 char:23
+ $a | move-itemproperty <<<<  -path hklm:\software\mycompany\design -name product
```

To investigate, use the Trace-Command cmdlet to trace the Parameter Binding component of Windows PowerShell. The following command traces the Parameter Binding component while the command is processing. It uses the -pshost parameter to display the results at the console and the -filepath command to send them to the debug.txt file for later reference.

```
trace-command -name parameterbinding -expression {get-item -path
hklm:\software\mycompany\sales |
  move-itemproperty -path hklm:\software\mycompany\design -name product} -pshost -filepath
debug.txt
```

The results of the trace are lengthy, but they show the values being bound to the Get-Item cmdlet and then the named values being bound to the Move-ItemProperty cmdlet.

```
...
BIND NAMED cmd line args [Move-ItemProperty]
   BIND arg [hklm:\software\mycompany\design] to parameter [Path]
...
   BIND arg [product] to parameter [Name]
....
BIND POSITIONAL cmd line args [Move-ItemProperty]
...
```

Finally, it shows that the attempt to bind the path to the Destination parameter of Move-ItemProperty failed.

```
...
BIND PIPELINE object to parameters: [Move-ItemProperty]
   PIPELINE object TYPE = [Microsoft.Win32.RegistryKey]
```

RESTORING pipeline parameter's original values
Parameter [Destination] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION
Parameter [Credential] PIPELINE INPUT ValueFromPipelineByPropertyName NO COERCION
...

To investigate the failure, use the Get-Help cmdlet to view the attributes of the Destination parameter. The following command gets detailed information about the Destination parameter.

```
get-help move-itemproperty -parameter destination
```

The results show that Destination takes pipeline input only "by property name". That is, the piped object must have a property named Destination.

-destination <string>
Specifies the path to the destination location.

Required? true
Position? 2
Default value
Accept pipeline input? true (ByPropertyName)
Accept wildcard characters? true

To see the properties of the object being piped to the Move-ItemProperty cmdlet, pipe it to the Get-Member cmdlet. The following command pipes the results of the first part of the command to the Get-Member cmdlet.

```
get-item -path hklm:\software\mycompany\sales | get-member
```

The output shows that the item is a Microsoft.Win32.RegistryKey that does not have a Destination property. That explains why the command failed.

To fix the command, we must specify the destination in the Move-ItemProperty cmdlet. We can use a Get-ItemProperty command to get the path, but the name and destination must be specified in the Move-ItemProperty part of the command.

```
get-item -path hklm:\software\mycompany\design |
  move-itemproperty -dest hklm:\software\mycompany\design -name product
```

To verify that the command worked, use a Get-ItemProperty command:

```
get-itemproperty hklm:\software\mycompany\sales
```

The results show that the Product registry entry was moved to the Sales key.

```
PSPath      :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software\mycompany\sales
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\software\mycompany
```

PSChildName : sales
PSDrive   : HKLM
PSProvider   : Microsoft.PowerShell.Core\Registry
Product   : 18

SEE ALSO
about_objects
about_parameters
about_command_syntax
about_foreach

TOPIC

about_PowerShell.exe

SHORT DESCRIPTION

Explains how to use the PowerShell.exe command-line tool. Displays the syntax and describes the command-line switches.

PowerShell.exe starts a Windows PowerShell session. You can use it in Cmd.exe and in Windows PowerShell.

LONG DESCRIPTION

SYNTAX

```
PowerShell[.exe]
  [-EncodedCommand <Base64EncodedCommand>]
  [-ExecutionPolicy <ExecutionPolicy>]
  [-InputFormat {Text | XML}]
  [-Mta]
  [-NoExit]
  [-NoLogo]
  [-NonInteractive]
  [-NoProfile]
  [-OutputFormat {Text | XML}]
  [-PSConsoleFile <FilePath> | -Version <Windows PowerShell version>]
```

[-Sta]
[-WindowStyle <style>]
[-File <FilePath> [<Args>]]
[-Command { - | <script-block> [-args <arg-array>]
| <string> [<CommandParameters>] }]
PowerShell[.exe] -Help | -? | /?

PARAMETERS

-EncodedCommand <Base64EncodedCommand>
Accepts a base-64-encoded string version of a command.
Use this parameter to submit commands to Windows PowerShell
that require complex quotation marks or curly braces.

-ExecutionPolicy <ExecutionPolicy>
Sets the default execution policy for the current session
and saves it in the $env:PSExecutionPolicyPreference environment
variable. This parameter does not change the Windows PowerShell
execution policy that is set in the registry. For information
about Windows PowerShell execution policies, including a list
of valid values, see about_Execution_Policies
(http://go.microsoft.com/fwlink/?LinkID=135170).

-File <FilePath> [<Parameters>]
Runs the specified script in the local scope ("dot-sourced"),
so that the functions and variables that the script creates
are available in the current session. Enter the script file
path and any parameters. File must be the last parameter in
the command, because all characters typed after the File
parameter name are interpreted as the script file path followed
by the script parameters.

You can include the parameters of a script, and parameter
values, in the value of the File parameter. For example:
-File .\Get-Script.ps1 -Domain Central

Typically, the switch parameters of a script are either
included or omitted. For example, the following command
uses the All parameter of the Get-Script.ps1 script file:
-File .\Get-Script.ps1 -All

In rare cases, you might need to provide a Boolean value
for a switch parameter. To provide a Boolean value for a
switch parameter in the value of the File parameter,
enclose the parameter name and value in curly braces,
such as the following:
-File .\Get-Script.ps1 {-All:$False}.

-InputFormat {Text | XML}
Describes the format of data sent to Windows PowerShell. Valid values are "Text" (text strings) or "XML" (serialized CLIXML format).

-Mta
Starts Windows PowerShell using a multi-threaded apartment. This parameter is introduced in Windows PowerShell 3.0. In Windows PowerShell 2.0, multi-threaded apartment (MTA) is the default. In Windows PowerShell 3.0, single-threaded apartment (STA) is the default.

-NoExit
Does not exit after running startup commands.

-NoLogo
Hides the copyright banner at startup.

-NonInteractive
Does not present an interactive prompt to the user.

-NoProfile
Does not load the Windows PowerShell profile.

-OutputFormat {Text | XML}
Determines how output from Windows PowerShell is formatted. Valid values are "Text" (text strings) or "XML" (serialized CLIXML format).

-PSConsoleFile <FilePath>
Loads the specified Windows PowerShell console file. Enter the path and name of the console file. To create a console file, use the Export-Console cmdlet in Windows PowerShell.

-Sta
Starts Windows PowerShell using a single-threaded apartment. In Windows PowerShell 2.0, multi-threaded apartment (MTA) is the default. In Windows PowerShell 3.0, single-threaded apartment (STA) is the default.

-Version <Windows PowerShell Version>
Starts the specified version of Windows PowerShell. Valid values are 2.0 and 3.0. The version that you specify must be installed on the system. If Windows PowerShell 3.0 is installed on the computer, "3.0" is the default version. Otherwise, "2.0" is the default version. For more information, see "Installing Windows PowerShell" in the Windows PowerShell Getting Started Guide.

-WindowStyle <Window style>
Sets the window style for the session. Valid values are Normal, Minimized, Maximized and Hidden.

-Command
Executes the specified commands (and any parameters) as though they were typed at the Windows PowerShell command prompt, and then exits, unless the NoExit parameter is specified.

The value of Command can be "-", a string. or a script block. If the value of Command is "-", the command text is read from standard input.

Script blocks must be enclosed in braces ({}). You can specify a script block only when running PowerShell.exe in Windows PowerShell. The results of the script are returned to the parent shell as deserialized XML objects, not live objects.

If the value of Command is a string, Command must be the last parameter in the command , because any characters typed after the command are interpreted as the command arguments.

To write a string that runs a Windows PowerShell command, use the format:
"& {<command>}"
where the quotation marks indicate a string and the invoke operator (&) causes the command to be executed.

-Help, -?, /?
Displays help for PowerShell.exe. If you are typing a PowerShell.exe command in Windows PowerShell, prepend the command parameters with a hyphen (-), not a forward slash (/). You can use either a hyphen or forward slash in Cmd.exe.

REMARKS:
Troubleshooting note: In Windows PowerShell 2.0, starting some programs from the Windows PowerShell console fails with a LastExitCode of 0xc0000142.

EXAMPLES
PowerShell -PSConsoleFile sqlsnapin.psc1

PowerShell -Version 2.0 -NoLogo -InputFormat text -OutputFormat XML

PowerShell -Command {Get-EventLog -LogName security}

```
    PowerShell -Command "& {Get-EventLog -LogName security}"

# To use the -EncodedCommand parameter:
    $command = "dir 'c:\program files' "
    $bytes = [System.Text.Encoding]::Unicode.GetBytes($command)
    $encodedCommand = [Convert]::ToBase64String($bytes)
    powershell.exe -encodedCommand $encodedCommand


 SEE ALSO
    about_PowerShell_Ise.exe (http://go.microsoft.com/fwlink/?LinkID=256512)
```

TOPIC

about_PowerShell_Ise.exe

SHORT DESCRIPTION

Explains how to use the PowerShell_Ise.exe command-line tool.

LONG DESCRIPTION

PowerShell_Ise.exe starts a Windows PowerShell Integrated Scripting Environment (ISE) session. You can run it in Cmd.exe and in Windows PowerShell.

To run PowerShell_ISE.exe, type PowerShell_ISE.exe, PowerShell_ISE, or ISE.

SYNTAX

```
    PowerShell_Ise[.exe]
    PowerShell_ISE[.exe]
    ISE[.exe]
        [–File]<FilePath[]> [–NoProfile] [–MTA]
        –Help | ? | -? | /?
```

Displays the syntax and describes the command-line switches.

PARAMETERS

-File

Opens the specified files in Windows PowerShell ISE. The parameter name ("-File") is optional. To list more than one file, enter one text string enclosed in quotation marks. Use commas to separate the file names within the string.

For example:

```
PowerShell_ISE -File "File1.ps1,File2.ps1,File3.xml".
```

Spaces between the file names are permitted in Windows PowerShell, but might not be interpreted correctly by other programs, such as Cmd.exe.

You can use this parameter to open any text file, including Windows PowerShell script files and XML files.

-Mta

Starts Windows PowerShell ISE using a multi-threaded apartment. This parameter is introduced in Windows PowerShell 3.0. Single-threaded apartment (STA) is the default.

-NoProfile

Does not run Windows PowerShell profiles. By default, Windows PowerShell profiles are run in every session.

This parameter is recommended when you are writing shared content, such as functions and scripts that will be run on systems with different profiles. For more information, see about_Profiles (http://go.microsoft.com/fwlink/?LinkID=113729).

-Help, -?, /?

Displays help for PowerShell_ISE.exe.

EXAMPLES

These commands start Windows PowerShell ISE. The commands are equivalent and can be used interchangeably.

```
PS C:\>PowerShell_ISE.exe
PS C:\>PowerShell_ISE
PS C:\>ISE
```

These commands open the Get-Profile.ps1 script in Windows PowerShell ISE. The commands are equivalent and can be used interchangeably.

```
PS C:\>PowerShell_ISE.exe -File .\Get-Profile.ps1
```

```
PS C:\>ISE -File .\Get-Profile.ps1
PS C:\>ISE .\Get-Profile.ps1
```

This command opens the Get-Backups.ps1 and Get-BackupInstance.ps1 scripts in Windows PowerShell ISE. To open more than one file, use a comma to separate the file names and enclose the entire file name value in quotation marks.

```
PS C:\>ISE -File ".\Get-Backups.ps1,Get-BackupInstance.ps1"
```

This command starts Windows PowerShell ISE with no profiles.

```
PS C:\>ISE -NoProfile
```

This command gets help for PowerShell_ISE.exe.

```
PS C:\>ISE -help
```

SEE ALSO

about_PowerShell.exe
about_Windows_PowerShell_ISE
Windows PowerShell 3.0 Integrated Scripting Environment (ISE)

TOPIC

Preference Variables

SHORT DESCRIPTION

Variables that customize the behavior of Windows PowerShell

LONG DESCRIPTION

Windows PowerShell includes a set of variables that enable you to customize its behavior. These "preference variables" work like the

options in GUI-based systems.

The preference variables affect the Windows PowerShell operating environment and all commands run in the environment. In many cases, the cmdlets have parameters that you can use to override the preference behavior for a specific command.

The following table lists the preference variables and their default values.

Variable	Default Value
$ConfirmPreference	High
$DebugPreference	SilentlyContinue
$ErrorActionPreference	Continue
$ErrorView	NormalView
$FormatEnumerationLimit	4
$InformationPreference	SilentlyContinue
$LogCommandHealthEvent	False (not logged)
$LogCommandLifecycleEvent	False (not logged)
$LogEngineHealthEvent	True (logged)
$LogEngineLifecycleEvent	True (logged)
$LogProviderLifecycleEvent	True (logged)
$LogProviderHealthEvent	True (logged)
$MaximumAliasCount	4096
$MaximumDriveCount	4096
$MaximumErrorCount	256
$MaximumFunctionCount	4096
$MaximumHistoryCount	4096
$MaximumVariableCount	4096
$OFS	(Space character (" "))
$OutputEncoding	ASCIIEncoding object
$ProgressPreference	Continue
$PSDefaultParameterValues	(None - empty hash table)
$PSEmailServer	(None)
$PSModuleAutoLoadingPreference	All
$PSSessionApplicationName	WSMAN
$PSSessionConfigurationName	http://schemas.microsoft.com/PowerShell/microsoft.PowerShell
$PSSessionOption	(See below)
$VerbosePreference	SilentlyContinue
$WarningPreference	Continue
$WhatIfPreference	0

Windows PowerShell also includes the following environment variables that store user preferences. For more information about these environment variables, see about_Environment_Variables.

Variable

PSExecutionPolicyPreference
PSModulePath

WORKING WITH PREFERENCE VARIABLES

This document describes each of the preference variables.

To display the current value of a specific preference variable, type the name of the variable. In response, Windows PowerShell provides the value. For example, the following command displays the value of the $ConfirmPreference variable.

```
PS> $ConfirmPreference
High
```

To change the value of a variable, use an assignment statement. For example, the following statement assigns the value "Medium" to the $ConfirmPreference variable.

```
PS> $ConfirmPreference = "Medium"
```

Like all variables, the values that you set are specific to the current Windows PowerShell session. To make them effective in all Windows PowerShell session, add them to your Windows PowerShell profile. For more information, see about_Profiles.

WORKING REMOTELY

When you run commands on a remote computer, the remote commands are subject only to the preferences set in the Windows PowerShell client on the remote computer. For example, when you run a remote command, the value of the $DebugPreference variable on remote computer determines how Windows PowerShell responds to debugging messages.

For more information about remote commands, see about_remote.

$ConfirmPreference

Determines whether Windows PowerShell automatically prompts you for confirmation before running a cmdlet or function.

When the value of the $ConfirmPreference variable (High, Medium, Low) is less than or equal to the risk assigned to the cmdlet or function (High,

Medium, Low), Windows PowerShell automatically prompts you for confirmation before running the cmdlet or function.

If the value of the $ConfirmPreference variable is None, Windows PowerShell never automatically prompts you before running a cmdlet or function.

To change the confirming behavior for all cmdlets and functions in the session, change the value of the $ConfirmPreference variable.

To override the $ConfirmPreference for a single command, use the Confirm parameter of the cmdlet or function. To request confirmation, use -Confirm. To suppress confirmation, use -Confirm:$false

Valid values of $ConfirmPreference:

None: Windows PowerShell does not prompt automatically.
To request confirmation of a particular command, use the Confirm parameter of the cmdlet or function.

Low: Windows PowerShell prompts for confirmation before running cmdlets or functions with a low, medium, or high risk.

Medium: Windows PowerShell prompts for confirmation before running cmdlets or functions with a medium, or high risk.

High: Windows PowerShell prompts for confirmation before running cmdlets or functions with a high risk.

DETAILED EXPLANATION

When the actions of a cmdlet or function significantly affect the system, such as those that delete data or use a significant amount of system resources, Windows PowerShell can automatically prompt you for confirmation before performing the action.

For example,

```
PS> remove-item file.txt

    Confirm
    Are you sure you want to perform this action?
    Performing operation "Remove File" on Target "C:\file.txt".
    [Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):
```

The estimate of the risk is an attribute of the cmdlet or function known as its "ConfirmImpact". Users cannot change it.

Cmdlets and functions that might pose a risk to the system have a Confirm parameter that you can use to request or suppress confirmation for a single command.

Because most cmdlets and functions use the default risk value (ConfirmImpact) of Medium, and the default value of $ConfirmPreference is High, automatic confirmation rarely occurs. However, you can activate automatic confirmation by changing the value of $ConfirmPreference to Medium or Low.

EXAMPLES

This example shows the effect of the default value of $ConfirmPreference. The High value only confirms high-risk cmdlets and functions. Since most cmdlets and functions are medium risk, they are not automatically confirmed.

```
PS> $confirmpreference            #Get the current value of the
High                              variable

PS> remove-item temp1.txt         #Delete a file
PS>                               #Deleted without confirmation


PS> remove-item temp2.txt -confirm  #Use the Confirm parameter to
                                    request confirmation

Confirm
Are you sure you want to perform this action?
Performing operation "Remove File" on Target "C:\temp2.txt".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):
```

The following example shows the effect of changing the value of $ConfirmPreference to Medium. Because most cmdlets and function are medium-risk, they are automatically confirmed. To suppress the confirmation prompt for a single command, use the Confirm parameter with a value of $false

```
PS> $confirmpreference = "Medium"  #Change the value of $ConfirmPreference
PS> remove-item temp2.txt         #Deleting a file triggers confirmation

Confirm
Are you sure you want to perform this action?
Performing operation "Remove File" on Target "C:\temp2.txt".
```

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):

```
PS> remove-item temp3.txt -confirm:$false   #Use Confirm parameter
                                to suppress confirmation
PS>
```

$DebugPreference

Determines how Windows PowerShell responds to debugging messages generated by a script, cmdlet or provider, or by a Write-Debug command at the command line.

Some cmdlets display debugging messages, which are typically very technical messages designed for programmers and technical support professionals. By default, debugging messages are not displayed, but you can display debugging messages by changing the value of $DebugPreference.

You can also use the Debug common parameter of a cmdlet to display or hide the debugging messages for a specific command. For more information, type: "get-help about_commonparameters".

Valid values:

Stop: Displays the debug message and stops executing. Writes an error to the console.

Inquire: Displays the debug message and asks you whether you want to continue. Note that adding the Debug common parameter to a command--when the command is configured to generate a debugging message--changes the value of the $DebugPreference variable to Inquire.

Continue: Displays the debug message and continues with execution.

SilentlyContinue: (Default) No effect. The debug message is not displayed and execution continues without interruption.

EXAMPLES

The following examples show the effect of changing the values of $DebugPreference when a Write-Debug command is entered at the command

line. The change affects all debugging messages, including those generated by cmdlets and scripts. The examples also show the use of the Debug common parameter, which displays or hides the debugging messages related to a single command.

This example shows the effect of the default value, "SilentlyContinue." The debug message is not displayed and processing continues. The final command uses the Debug parameter to override the preference for a single command.

```
PS> $debugpreference                # Get the current value of
SilentlyContinue                  $DebugPreference

PS> write-debug "Hello, World"
PS>                               # The debug message is not
                                  displayed.

PS> write-debug "Hello, World" -Debug   # Use the Debug parameter
DEBUG: Hello, World                     # The debug message is
                                  displayed and confirmation is requested.
Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default is "Y"):
```

This example shows the effect of the "Continue" value. The final command uses the Debug parameter with a value of $false to suppress the message for a single command.

```
PS> $debugpreference = "Continue"   # Change the value to "Continue"

PS> write-debug "Hello, World"
DEBUG: Hello, World               # The debug message is displayed
PS>                               and processing continues.


PS> write-debug "Hello, World" -Debug:$false
                                # Use the Debug parameter with
                                  false.
PS>                               # The debug message is not
                                  displayed.
```

This example shows the effect of the "Stop" value. The final command uses the Debug parameter with a value of $false to suppress the message for a single command.

```
PS> $debugpreference = "Stop"       #Change the value to "Stop"
```

```
PS> write-debug "Hello, World"
DEBUG: Hello, World
Write-Debug : Command execution stopped because the shell variable "DebugPreference" is
set to Stop.
At line:1 char:12
+ write-debug  <<<< "Hello, World"

PS> write-debug "Hello, World" -Debug:$false
                          # Use the Debug parameter with
                            $false
PS>                        # The debug message is not
                          displayed and processing is
                          not stopped.
```

This example shows the effect of the "Inquire" value. The final command uses the Debug parameter with a value of $false to suppress the message for a single command.

```
PS> $debugpreference = "Inquire"
PS> write-debug "Hello, World"
DEBUG: Hello, World

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default is "Y"):

PS> write-debug "Hello, World" -Debug:$false
                          # Use the Debug parameter with
                            $false
PS>                        # The debug message is not
                          displayed and processing
                          continues without interruption.
```

$ErrorActionPreference

Determines how Windows PowerShell responds to a non-terminating error (an error that does not stop the cmdlet processing) at the command line or in a script, cmdlet, or provider, such as the errors generated by the Write-Error cmdlet.

You can also use the ErrorAction common parameter of a cmdlet to override the preference for a specific command.

Valid values:

Stop: Displays the error message and stops executing.

Inquire: Displays the error message and asks you whether you want to continue.

Continue: (Default) Displays the error message and continues executing.

Suspend: Automatically suspends a workflow job to allow for further investigation. After investigation, the workflow can be resumed.

SilentlyContinue: No effect. The error message is not displayed and execution continues without interruption.

NOTE: The Ignore value of the ErrorAction common parameter is not a valid value of the $ErrorActionPreference variable. The Ignore value is intended for per-command use, not for use as saved preference.

Neither $ErrorActionPreference nor the ErrorAction common parameter affect how Windows PowerShell responds to terminating errors (those that stop cmdlet processing).

For more information about the ErrorAction common parameter, see about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

EXAMPLES

These examples show the effect of the different values of $ErrorActionPreference and the use of the ErrorAction common parameter to override the preference for a single command. The ErrorAction parameter has the same valid values as the $ErrorActionPreference variable.

This example shows the effect of the Continue value, which is the default.

```
PS> $erroractionpreference
Continue# Display the value of the preference.

PS> write-error "Hello, World"
                # Generate a non-terminating error.

write-error "Hello, World" : Hello, World
                # The error message is displayed and
                  execution continues.
```

```
PS> write-error "Hello, World" -ErrorAction:SilentlyContinue
                # Use the ErrorAction parameter with a
                  value of "SilentlyContinue".
PS>
                # The error message is not displayed and
                  execution continues.
```

This example shows the effect of the SilentlyContinue value.

```
PS> $ErrorActionPreference = "SilentlyContinue"
                # Change the value of the preference.
PS> write-error "Hello, World"
                # Generate an error message.
PS>
                # Error message is suppressed.
PS> write-error "Hello, World" -erroraction:continue
                # Use the ErrorAction parameter with a
                  value of "Continue".
write-error "Hello, World" -erroraction:continue : Hello, World
                # The error message is displayed and
                  execution continues.
```

This example shows the effect of a real error. In this case, the command gets a non-existent file, nofile.txt. The example also uses the ErrorAction common parameter to override the preference.

```
PS> $erroractionpreference
SilentlyContinue       # Display the value of the preference.


PS> get-childitem -path nofile.txt
PS>                 # Error message is suppressed.

PS> $ErrorActionPreference = "Continue"
                # Change the value to Continue.

PS> get-childitem -path nofile.txt
Get-ChildItem : Cannot find path 'C:\nofile.txt' because it does not exist.
At line:1 char:4
+ get-childitem  <<<< nofile.txt

PS> get-childitem -path nofile.txt -erroraction SilentlyContinue
                # Use the ErrorAction parameter
PS>
                # Error message is suppressed.
```

```
PS> $ErrorActionPreference = "Inquire"
                # Change the value to Inquire.
PS> get-childitem -path nofile.txt

Confirm
Cannot find path 'C:\nofile.txt' because it does not exist.
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default is "Y"): y

Get-ChildItem : Cannot find path 'C:\nofile.txt' because it does not exist.
At line:1 char:4
+ get-childitem  <<<< nofile.txt

PS> $ErrorActionPreference = "Continue"
                # Change the value to Continue.
PS> Get-Childitem nofile.txt -erroraction "Inquire"
                # Use the ErrorAction parameter to override
                  the preference value.

Confirm
Cannot find path 'C:\nofile.txt' because it does not exist.
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default is "Y"):
```

$ErrorView

Determines the display format of error messages in Windows PowerShell.

Valid values:

NormalView: A detailed view designed for most users.
(default) Consists of a description of the error, the name of the object involved in the error, and arrows (<<<<) that point to the words in the command that caused the error.

CategoryView: A succinct, structured view designed for production environments. The format is:
{Category}: ({TargetName}:{TargetType}):[{Activity}], {Reason}

For more information about the fields in CategoryView, see "ErrorCategoryInfo class" in the Windows PowerShell SDK.

EXAMPLES

These example show the effect of the ErrorView values.

This example shows how an error appears when the value of $ErrorView is NormalView. In this case, the Get-ChildItem command is used to find a

non-existent file.

```
PS> $ErrorView                    # Verify the value.
NormalView

PS> get-childitem nofile.txt          # Find a non-existent file.
Get-ChildItem : Cannot find path 'C:\nofile.txt' because it does not exist.
At line:1 char:14
+ get-childitem  <<<< nofile.txt
```

This example shows how the same error appears when the value of $ErrorView is CategoryView.

```
PS> $ErrorView = "CategoryView"       # Change the value to
                              CategoryView

PS> get-childitem nofile.txt
ObjectNotFound: (C:\nofile.txt:String) [Get-ChildItem], ItemNotFoundException
```

This example demonstrates that the value of ErrorView only affects the error display; it does not change the structure of the error object that is stored in the $error automatic variable. For information about the $error automatic variable, see about_automatic_variables.

This command takes the ErrorRecord object associated with the most recent error in the error array (element 0) and formats all of the properties of the error object in a list.

```
PS> $error[0] | format-list -property * -force

Exception    : System.Management.Automation.ItemNotFoundException: Cannot find path
           'C:\nofile.txt' because it does not exist.
           at System.Management.Automation.SessionStateInternal.GetChildItems(String path,
           Boolean recurse, CmdletProviderContext context)
           at System.Management.Automation.ChildItemCmdletProviderIntrinsics.Get(String path,
           Boolean recurse, CmdletProviderContext context)
           at Microsoft.PowerShell.Commands.GetChildItemCommand.ProcessRecord()
TargetObject        : C:\nofile.txt
CategoryInfo        : ObjectNotFound: (C:\nofile.txt:String) [Get-ChildItem],
                  ItemNotFoundException
FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand
ErrorDetails        :
InvocationInfo      : System.Management.Automation.InvocationInfo
```

$FormatEnumerationLimit

Determines how many enumerated items are included in a display. This variable does not affect the underlying objects; just the display. When the value of $FormatEnumerationLimit is less than the number of enumerated items, Windows PowerShell adds an ellipsis (...) to indicate items not shown.

Valid values: Integers (Int32)
Default value: 4

EXAMPLES

This example shows how to use the $FormatEnumerationLimit variable to improve the display of enumerated items.

The command in this example generates a table that lists all of the services running on the computer in two groups; one for running services and one for stopped services. It uses a Get-Service command to get all of the services, and then send the results through the pipeline to the Group-Object cmdlet, which groups the results by the service status.

The resulting display is a table that lists the status in the Name column and the processes with that status in the Group column. (To change the column labels, use a hash table. For more information, see the examples in "get-help format-table -examples".)

There are a maximum of 4 services listed in the Group column for each status. To increase the number of items listed, increase the value of $FormatEnumerationLimit to 1000.

In the resulting display, the list in the Group column is now limited by the line length. In the final command in the example, use the Wrap parameter of Format-Table to display all of the processes in each Status group.

```
PS> $formatenumerationlimit        # Find the current value
4

PS> get-service | group-object -property status
                          # List all services grouped by
                            status

Count Name                      Group
----- ----                     -----
   60 Running                  {AdtAgent, ALG, Ati HotKey Poller, AudioSrv...}
   41 Stopped                  {Alerter, AppMgmt, aspnet_state, ATI Smart...}
```

```
                        # The list is truncated after
                          4 items.


PS> $formatenumerationlimit = 1000
                        # Increase the limit to 1000.

PS> get-service | group-object -property status
                        # Repeat the command.

Count Name     Group
----- ----     -----
   60 Running  {AdtAgent, ALG, Ati HotKey Poller, AudioSrv, BITS, CcmExec...
   41 Stopped  {Alerter, AppMgmt, aspnet_state, ATI Smart, Browser, CiSvc...


PS> get-service | group-object -property status | format-table -wrap
                        # Add the Wrap parameter.

Count Name      Group
----- ----      -----
   60 Running   {AdtAgent, ALG, Ati HotKey Poller, AudioSrv, BITS, CcmExec, Client
                for NFS, CryptSvc, DcomLaunch, Dhcp, dmserver, Dnscache, ERSvc,
                Eventlog, EventSystem, FwcAgent, helpsvc, HidServ, IISADMIN,
                InoRPC, InoRT, InoTask, lanmanserver, lanmanworkstation, LmHosts,
                MDM, Netlogon, Netman, Nla, NtLmSsp, PlugPlay, PolicyAgent,
                ProtectedStorage, RasMan, RemoteRegistry, RpcSs, SamSs, Schedule,
                seclogon, SENS, SharedAccess, ShellHWDetection, SMT PSVC, Spooler,
                srservice, SSDPSRV, stisvc, TapiSrv, TermService, Themes, TrkWks,
                UMWdf, W32Time, W3SVC, WebClient, winmgmt, wscsvc, wuauserv,
                WZCSVC, zzInterix}

   41 Stopped   {Alerter, AppMgmt, aspnet_state, ATI Smart, Browser, CiSvc,
                ClipSrv, clr_optimization_v2.0.50727_32, COMSysApp, CronService,
                dmadmin, FastUserSwitchingCompatibility, HTTPFilter, ImapiService,
                Mapsvc, Messenger, mnmsrvc, MSDTC, MSIServer, msvsmon80, NetDDE,
                NetDDEdsdm, NtmsSvc, NVSvc, ose, RasAuto, RDSessMgr, RemoteAccess,
                RpcLocator, SCardSvr, SwPrv, SysmonLog, TlntSvr, upnphost, UPS,
                VSS, WmdmPmSN, Wmi, WmiApSrv, xmlprov}
```

$InformationPreference

The $InformationPreference variable lets you set information stream preferences (specifically, informational messages that you have added to commands or scripts by adding the Write-Information cmdlet, and want displayed to users) for a Windows PowerShell session. The value of the -InformationAction parameter, if used, overrides the current value of

the $InformationPreference variable.

Valid values:

Stop: Stops a command or script at an occurrence of the Write-Information command.

Inquire: Displays the informational message that you specify in a Write-Information command, then asks whether you want to continue.

Continue: Displays the informational message, and continues running.

Suspend: Automatically suspends a workflow job after a Write-Information command is carried out, to allow users to see the messages before continuing. The workflow can be resumed at the user's discretion.

SilentlyContinue: (Default) No effect. The informational messages are not displayed, and the script continues without interruption.

$Log*Event

The Log*Event preference variables determine which types of events are written to the Windows PowerShell event log in Event Viewer. By default, only engine and provider events are logged, but you can use the Log*Event preference variables to customize your log, such as logging events about commands.

The Log*Event preference variables are as follows:

$LogCommandHealthEvent: Logs errors and exceptions in command initialization and processing. Default = $false (not logged).

$LogCommandLifecycleEvent:
Logs the starting and stopping of commands and command pipelines and security exceptions in command discovery. Default = $false (not logged).

$LogEngineHealthEvent: Logs errors and failures of sessions. Default = $true (logged).

$LogEngineLifecycleEvent: Logs the opening and closing of sessions. Default = $true (logged).

$LogProviderHealthEvent: Logs provider errors, such as read and write errors, lookup errors, and invocation errors. Default = $true (logged).

$LogProviderLifecycleEvent: Logs adding and removing of Windows PowerShell providers. Default = $true (logged). (For information about Windows PowerShell providers, type: "get-help about_provider".

To enable a Log*Event, type the variable with a value of $true, for example:

$LogCommandLifeCycleEvent

- or -

$LogCommandLifeCycleEvent = $true

To disable an event type, type the variable with a value of $false, for example:

$LogCommandLifeCycleEvent = $false

The events that you enable are effective only for the current Windows PowerShell console. To apply the configuration to all consoles, save the variable settings in your Windows PowerShell profile.

$MaximumAliasCount

Determines how many aliases are permitted in a Windows PowerShell session. The default value, 4096, should be sufficient for most uses, but you can adjust it to meet your needs.

Valid values: 1024 - 32768 (Int32)
Default: 4096

To count the aliases on your system, type:

(get-alias).count

$MaximumDriveCount

Determines how many Windows PowerShell drives are permitted in a given session. This includes file system drives and data stores that are exposed by Windows PowerShell providers and appear as drives, such as the Alias: and HKLM: drives.

Valid values: 1024 - 32768 (Int32)
Default: 4096

To count the aliases on your system, type:

(get-psdrive).count

$MaximumErrorCount

Determines how many errors are saved in the error history for the session.

Valid values: 256 - 32768 (Int32)
Default: 256

Objects that represent each retained error are stored in the $Error automatic variable. This variable contains an array of error record objects, one for each error. The most recent error is the first object in the array ($Error[0]).

To count the errors on your system, use the Count property of the $Error array. Type:

$Error.count

To display a specific error, use array notation to display the error. For example, to see the most recent error, type:

```
$Error[0]
```

To display the oldest retained error, type:

```
$Error[($Error.Count -1]
```

To display the properties of the ErrorRecord object, type:

```
$Error[0] | format-list -property * -force
```

In this command, the Force parameter overrides the special formatting of ErrorRecord objects and reverts to the conventional format.

To delete all errors from the error history, use the Clear method of the error array.

```
PS> $Error.count
17
PS> $Error.clear()
PS>
PS> $Error.count
```

0

To find all properties and methods of an error array, use the Get-Member cmdlet with its InputObject parameter. When you pipe a collection of objects to Get-Member, Get-Member displays the properties and methods of the objects in the collection. When you use the InputObject parameter of Get-Member, Get-Member displays the properties and methods of the collection.

$MaximumFunctionCount

Determines how many functions are permitted in a given session.

Valid values: 1024 - 32768 (Int32)
Default: 4096

To see the functions in your session, use the Windows PowerShell Function: drive that is exposed by the Windows PowerShell Function provider. (For more information about the Function provider, type "get-help function").

To list the functions in the current session, type:

```
get-childitem function:
```

To count the functions in the current session, type:

```
(get-childitem function:).count
```

$MaximumHistoryCount

Determines how many commands are saved in the command history for the current session.

Valid values: 1 - 32768 (Int32)
Default: 4096

To determine the number of commands current saved in the command history, type:

```
(get-history).count
```

To see the command saved in your session history, use the Get-History cmdlet. For more information, see about_History (http://go.microsoft.com/fwlink/?LinkID=113233).

NOTE: In Windows PowerShell 2.0, the default value of the
$MaximumHistoryCount variable is 64.

$MaximumVariableCount

Determines how many variables are permitted in a given session, including automatic variables, preference variables, and the variables that you create in commands and scripts.

Valid values: 1024 - 32768 (Int32)
Default: 4096

To see the variables in your session, use the Get-Variable cmdlet and the features of the Windows PowerShell Variable: drive and the Windows PowerShell Variable provider. For information about the Variable provider, type "get-help variable".

To find the current number of variables on the system, type:

```
(get-variable).count
```

$OFS

Output Field Separator. Specifies the character that separates the elements of an array when the array is converted to a string.

Valid values: Any string.
Default: Space

By default, the $OFS variable does not exist and the output file separator is a space, but you can add this variable and set it to any string.

EXAMPLES

This example shows that a space is used to separate the values when an array is converted to a string. In this case, an array of integers is stored in a variable and then the variable is cast as a string.

```
PS> $array = 1,2,3              # Store an array of integers.

PS> [string]$array              # Cast the array to a string.
1 2 3                       # Spaces separate the elements
```

To change the separator, add the $OFS variable by assigning a value

to it. To work correctly, the variable must be named $OFS.

```
PS> $OFS = "+"                # Create $OFS and assign a "+"

PS> [string]$array            # Repeat the command
1+2+3                         # Plus signs separate the elements
```

To restore the default behavior, you can assign a space (" ") to the value of $OFS or delete the variable. This command deletes the variable and then verifies that the separator is a space.

```
PS> Remove-Variable OFS       # Delete $OFS
PS>

PS> [string]$array            # Repeat the command
1 2 3                         # Spaces separate the elements
```

$OutputEncoding

Determines the character encoding method that Windows PowerShell uses when it sends text to other applications.

For example, if an application returns Unicode strings to Windows PowerShell, you might need to change the value to UnicodeEncoding to send the characters correctly.

Valid values: Objects derived from an Encoding class, such as ASCIIEncoding, SBCSCodePageEncoding, UTF7Encoding, UTF8Encoding, UTF32Encoding, and UnicodeEncoding.

Default: ASCIIEncoding object (System.Text.ASCIIEncoding)

EXAMPLES

This example shows how to make the FINDSTR command in Windows work in Windows PowerShell on a computer that is localized for a language that uses Unicode characters, such as Chinese.

The first command finds the value of $OutputEncoding. Because the value is an encoding object, display only its EncodingName property.

```
PS> $OutputEncoding.EncodingName  # Find the current value
US-ASCII
```

In this example, a FINDSTR command is used to search for two Chinese characters that are present in the Test.txt file. When this FINDSTR

command is run in the Windows Command Prompt (Cmd.exe), FINDSTR finds the characters in the text file. However, when you run the same FINDSTR command in Windows PowerShell, the characters are not found because the Windows PowerShell sends them to FINDSTR in ASCII text, instead of in Unicode text.

```
PS> findstr <Unicode-characters>  # Use findstr to search.
PS>                               # None found.
```

To make the command work in Windows PowerShell, set the value of $OutputEncoding to the value of the OutputEncoding property of the console, which is based on the locale selected for Windows. Because OutputEncoding is a static property of the console, use double-colons (::) in the command.

```
PS> $OutputEncoding = [console]::outputencoding
PS>                               # Set the value equal to the
                                  OutputEncoding property of the
                                  console.
PS> $OutputEncoding.EncodingName
OEM United States
                                # Find the resulting value.
```

As a result of this change, the FINDSTR command finds the characters.

```
PS> findstr <Unicode-characters>
test.txt:       <Unicode-characters>

# Use findstr to search. It find the
                                characters in the text file.
```

$ProgressPreference

Determines how Windows PowerShell responds to progress updates generated by a script, cmdlet or provider, such as the progress bars generated by the Write-Progress cmdlet. The Write-Progress cmdlet creates progress bars that depict the status of a command.

Valid values:

Stop: Does not display the progress bar. Instead, it displays an error message and stops executing.

Inquire: Does not display the progress bar. Prompts for permission to continue. If you reply with Y or A, it displays the progress bar.

Continue: Displays the progress bar and continues with
(Default) execution.

SilentlyContinue: Executes the command, but does not display
the progress bar.

$PSEmailServer

Specifies the default e-mail server that is used to send e-mail messages. This preference variable is used by cmdlets that send e-mail, such as the Send-MailMessage cmdlet.

$PSDefaultParameterValues

Specifies default values for the parameters of cmdlets and advanced functions. The value of $PSDefaultParameterValues is a hash table where the key consists of the cmdlet name and parameter name separated by a colon (:) and the value is a custom default value that you specify.

This variable was introduced in Windows PowerShell 3.0

For more information about this preference variable, see about_Parameters_Default_Values.

$PSModuleAutoloadingPreference

Enables and disables automatic importing of modules in the session. "All" is the default. Regardless of the value of this variable, you can use the Import-Module cmdlet to import a module.

Valid values are:

All Modules are imported automatically on first-use. To import a module, get (Get-Command) or use any command in the module.

ModuleQualified
Modules are imported automatically only when a user uses the module-qualified name of a command in the module. For example, if the user types "MyModule\MyCommand", Windows PowerShell imports the MyModule module.

None Automatic importing of modules is disabled in the session. To import a module, use the Import-Module cmdlet.

For more information about automatic importing of modules, see about_Modules (http://go.microsoft.com/fwlink/?LinkID=144311).

$PSSessionApplicationName

Specifies the default application name for a remote command that uses WS-Management technology.

The system default application name is WSMAN, but you can use this preference variable to change the default.

The application name is the last node in a connection URI. For example, the application name in the following sample URI is WSMAN.

http://Server01:8080/WSMAN

The default application name is used when the remote command does not specify a connection URI or an application name.

The WinRM service uses the application name to select a listener to service the connection request. The value of this parameter should match the value of the URLPrefix property of a listener on the remote computer.

To override the system default and the value of this variable, and select a different application name for a particular session, use the ConnectionURI or ApplicationName parameters of the New-PSSession, Enter-PSSession or Invoke-Command cmdlets.

This preference variable is set on the local computer, but it specifies a listener on the remote computer. If the application name that you specify does not exist on the remote computer, the command to establish the session fails.

$PSSessionConfigurationName

Specifies the default session configuration that is used for PSSessions created in the current session.

This preference variable is set on the local computer, but it specifies a session configuration that is located on the remote computer.

The value of the $PSSessionConfigurationName variable is a fully qualified resource URI.

The default value:

http://schemas.microsoft.com/PowerShell/microsoft.PowerShell

indicates the Microsoft.PowerShell session configuration on the remote computer.

If you specify only a configuration name, the following schema URI is prepended:

http://schemas.microsoft.com/PowerShell/

You can override the default and select a different session configuration for a particular session by using the ConfigurationName parameter of the New-PSSession, Enter-PSSession or Invoke-Command cmdlets.

You can change the value of this variable at any time. When you do, remember that the session configuration that you select must exist on the remote computer. If it does not, the command to create a session that uses the session configuration fails.

This preference variable does not determine which local session configurations are used when remote users create a session that connects to this computer. However, you can use the permissions for the local session configurations to determine which users may use them.

$PSSessionOption

Establishes the default values for advanced user options in a remote session. These option preferences override the system default values for session options.

The $PSSessionOption variable contains a PSSessionOption object (System.Management.Automation.Remoting.PSSessionObject). Each property of the object represents a session option. For example, the NoCompression property turns of data compression during the session.

By default, the $PSSessionOption variable contains a PSSessionOption object with the default values for all options, as shown below.

```
MaximumConnectionRedirectionCount : 5
NoCompression                     : False
NoMachineProfile                  : False
```

```
ProxyAccessType               : None
ProxyAuthentication           : Negotiate
ProxyCredential               :
SkipCACheck                   : False
SkipCNCheck                   : False
SkipRevocationCheck           : False
OperationTimeout              : 00:03:00
NoEncryption                  : False
UseUTF16                      : False
IncludePortInSPN              : False
OutputBufferingMode           : None
Culture                       :
UICulture                     :
MaximumReceivedDataSizePerCommand :
MaximumReceivedObjectSize     : 209715200
ApplicationArguments          :
OpenTimeout                   : 00:03:00
CancelTimeout                 : 00:01:00
IdleTimeout                   : -00:00:00.0010000
```

For descriptions of these options, see the help topic for the
New-PSSessionOption cmdlet.

To change the value of the $PSSessionOption preference variable, use
the New-PSSessionOption cmdlet to create a PSSessionOption object with
the option values you prefer. Save the output in a variable called
$PSSessionOption.

For example,

```
$PSSessionOption = New-PSSessionOption -NoCompression
```

To use the $PSSessionOption preference variable in every
Windows PowerShell session, add a New-PSSessionOption command
that creates the $PSSessionOption variable to your Windows
PowerShell profile.

You can also set custom options for a particular remote session.
The options that you set take precedence over the system defaults
and the value of the $PSSessionOption preference variable.

To set custom session options, use the New-PSSessionOption cmdlet
to create a PSSessionOption object. Then, use the PSSessionOption object
as the value of the SessionOption parameter in cmdlets that create a session,
such as New-PSSession, Enter-PSSession, and Invoke-Command.

For more information about the New-PSSessionOption cmdlet, see

the help topic for New-PSSessionOption. For more information about remote commands and sessions, see about_Remote and about_PSSessions. For more information about using a profile, see about_Profiles.

$VerbosePreference

Determines how Windows PowerShell responds to verbose messages generated by a script, cmdlet or provider, such as the messages generated by the Write-Verbose cmdlet. Typically, verbose messages describe the actions performed to execute a command.

By default, verbose messages are not displayed, but you can change this behavior by changing the value of $VerbosePreference.

You can also use the Verbose common parameter of a cmdlet to display or hide the verbose messages for a specific command. For more information, type: "get-help about_commonparameters".

Valid values:

Stop: Displays the verbose message and an error message and then stops executing.

Inquire: Displays the verbose message and then displays a prompt that asks you whether you want to continue.

Continue: Displays the verbose message and then continues with execution.

SilentlyContinue: Does not display the verbose message. Continues executing. (Default)

EXAMPLES

These examples show the effect of the different values of $VerbosePreference and the use of the Verbose common parameter to override the preference value.

This example shows the effect of the SilentlyContinue value, which is the default.

```
PS> $VerbosePreference          # Find the current value.
SilentlyContinue

PS> Write-Verbose "Verbose message test."
PS>                             # Write a verbose message.
```

```
                        # Message is not displayed.

PS> Write-Verbose "Verbose message test." -verbose
VERBOSE: Verbose message test.
                    # Use the Verbose parameter.
```

This example shows the effect of the Continue value.

```
PS> $VerbosePreference = "Continue"
                        # Change the value to Continue.
PS> Write-Verbose "Verbose message test."
                        # Write a verbose message.
VERBOSE: Verbose message test.
                        # Message is displayed.

PS> Write-Verbose "Verbose message test." -verbose:$false
                        # Use the Verbose parameter with
                          a value of $false.
PS>
                        # Message is not displayed.
```

This example shows the effect of the Stop value.

```
PS> $VerbosePreference = "Stop"
                        # Change the value to Stop.
PS> Write-Verbose "Verbose message test."
                        # Write a verbose message.
VERBOSE: Verbose message test.
Write-Verbose : Command execution stopped because the shell variable "VerbosePreference"
is set to Stop.
At line:1 char:14
+ Write-Verbose  <<<< "Verbose message test."

PS> Write-Verbose "Verbose message test." -verbose:$false
                        # Use the Verbose parameter with
                          a value of $false
PS>
                        # Message is not displayed.
```

This example shows the effect of the Inquire value.

```
PS> $VerbosePreference = "Inquire"
                       # Change the value to Inquire.
PS> Write-Verbose "Verbose message test."
VERBOSE: Verbose message test.
                        # Write a verbose message.
```

Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "Y"): y
PS>

PS> Write-Verbose "Verbose message test." -verbose:$false
                    # Use the Verbose parameter.
PS>
                    # Message is not displayed.

$WarningPreference

Determines how Windows PowerShell responds to warning messages generated by a script, cmdlet or provider, such as the messages generated by the Write-Warning cmdlet.

By default, warning messages are displayed and execution continues, but you can change this behavior by changing the value of $WarningPreference.

You can also use the WarningAction common parameter of a cmdlet to determine how Windows PowerShell responds to warnings from a particular command. For more information, type: "get-help about_commonparameters".

Valid values:

Stop: Displays the warning message and an error message and then stops executing.

Inquire: Displays the warning message and then prompts for permission to continue.

Continue: (Default) Displays the warning message and then continues executing.

SilentlyContinue: Does not display the warning message. Continues executing.

EXAMPLES

These examples show the effect of the different values of $WarningPreference and the use of the WarningAction common parameter to override the preference value.

This example shows the effect of the Continue value, which is the

default.

```
PS> $WarningPreference    # Find the current value.
Continue

                # Write a warning message.
PS> Write-Warning "This action can delete data."
WARNING: This action can delete data.

                # Use the WarningAction parameter to
                # suppress the warning for this command
PS> Write-Warning "This action can delete data." -warningaction silentlycontinue
```

This example shows the effect of the SilentlyContinue value.

```
PS> $WarningPreference = "SilentlyContinue"
                # Change the value to SilentlyContinue.

PS> Write-Warning "This action can delete data."
PS>                 # Write a warning message.


PS> Write-Warning "This action can delete data." -warningaction stop
                # Use the WarningAction parameter to stop
                # processing when this command generates a
                # warning.
WARNING: This action can delete data.
Write-Warning : Command execution stopped because the shell variable
"WarningPreference" is set to Stop.
At line:1 char:14
+ Write-Warning <<<<  "This action can delete data." -warningaction stop
```

This example shows the effect of the Inquire value.

```
PS> $WarningPreference = "Inquire"
                # Change the value to Inquire.
PS> Write-Warning "This action can delete data."
                # Write a warning message.
WARNING: This action can delete data.

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default is "Y"): y
PS>
```

```
PS> Write-Warning "This action can delete data." -warningaction silentlycontinue
PS>                     # Use the WarningAction parameter to change the
                      # response to a warning for the current command.
```

This example shows the effect of the Stop value.

```
PS> $WarningPreference = "Stop"
                    # Change the value to Stop.

PS> Write-Warning "This action can delete data."
                    # Write a warning message.
WARNING: This action can delete data.
Write-Warning : Command execution stopped because the shell variable
  "WarningPreference" is set to Stop.
At line:1 char:14
+ Write-Warning  <<<< "This action can delete data."


PS> Write-Warning "This action can delete data." -warningaction inquire
WARNING: This action can delete data.

Confirm
Continue with this operation?
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default is "Y"):
                    # Use the WarningAction parameter to change the
                    # response to a warning for the current command.
```

$WhatIfPreference

Determines whether WhatIf is automatically enabled for every command that supports it. When WhatIf is enabled, the cmdlet reports the expected effect of the command, but does not execute the command.

Valid values:

0: (Default) WhatIf is not automatically enabled. To enable it manually, use the WhatIf parameter of the command.

1: WhatIf is automatically enabled on any command that supports it. Users can use the WhatIf command with a value of False to disable it manually (WhatIf:$false).

DETAILED EXPLANATION

When a cmdlet supports WhatIf, the cmdlet reports the expected effect of the command, instead of executing the command. For example, instead of deleting the test.txt file in response to a Remove-Item command, Windows PowerShell reports what it would delete. A subsequent Get-Childitem command confirms that the file was not deleted.

```
PS> remove-item test.txt
What if: Performing operation "Remove-Item" on Target "Item:
  C:\test.txt
PS> get-childitem test.txt

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


Mode                LastWriteTime     Length     Name
----                -------------     ------     ----
-a---        7/29/2006   7:15 PM         84     test.txt
```

EXAMPLES

These examples show the effect of the different values of $WhatIfPreference. They also show how to use the WhatIf cmdlet parameter to override the preference value for a specific command.

This example shows the effect of the 0 (not enabled) value, which is the default.

```
PS> $whatifpreference
0                    # Check the current value.

PS> get-childitem test.txt | format-list FullName
FullName : C:\test.txt
                     # Verify that the file exists.

PS> remove-item test.txt
PS>                  # Delete the file.

PS> get-childitem test.txt | format-list -property FullName
                     # Verify that the file is deleted.

Get-ChildItem : Cannot find path 'C:\test.txt' because it does not exist.
At line:1 char:14
```

```
+ get-childitem  <<<< test.txt | format-list fullname
```

This example shows the effect of using the WhatIf parameter when the value of $WhatIfPreference is 0.

```
PS> get-childitem test2.txt | format-list -property FullName
FullName : C:\test2.txt
                # Verify that the file exists.

PS> remove-item test2.txt -whatif
What if: Performing operation "Remove File" on Target "C:\test2.txt".
                # Use the WhatIf parameter

PS> get-childitem test2.txt | format-list -property FullName
FullName : C:\test2.txt
                # Verify that the file was not deleted
```

This example shows the effect of the 1 (WhatIf enabled) value. When you use Remove-Item to delete a cmdlet, Remove-Item displays the path to the file that it would delete, but it does not delete the file.

```
PS> $whatifpreference = 1
PS> $whatifpreference
1                # Change the value.

PS> remove-item test.txt
What if: Performing operation "Remove File" on Target "C:\test.txt".
                # Try to delete a file.

PS> get-childitem test.txt | format-list FullName
FullName : C:\test.txt
                # Verify that the file exists.
```

This example shows how to delete a file when the value of $WhatIfPreference is 1. It uses the WhatIf parameter with a value of $false.

```
PS> remove-item test.txt -whatif:$false
                # Use the WhatIf parameter with $false.
```

This example demonstrates that some cmdlets support WhatIf behavior and others do not. In this example, in which the value of $WhatIfPreference is 1 (enabled), a Get-Process command, which does not support WhatIf, is executed, but a Stop-Process command performs the WhatIf behavior. You can override the WhatIf behavior of the Stop-Process command by using the WhatIf parameter with a value of $false.

```
PS> $whatifpreference = 1
                # Change the value to 1.

PS> get-process winword
                # A Get-Process command completes.

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    234       8     6324      15060   154     0.36   2312 WINWORD


PS> stop-process -name winword
What if: Performing operation "Stop-Process" on Target "WINWORD (2312)".
                # A Stop-Process command uses WhatIf.

PS> stop-process -name winword  -whatif:$false
PS>                 # WhatIf:$false overrides the preference.

PS> get-process winword
Get-Process : Cannot find a process with the name 'winword'. Verify the process name
 and call the cmdlet again.
At line:1 char:12
+ get-process  <<<< winword
                # Verify that the process is stopped.
```

SEE ALSO

- about_Automatic_Variables
- about_CommonParameters
- about_Environment_Variables
- about_Profiles
- about_Remote
- about_Scopes
- about_Variables

TOPIC

about_Profiles

SHORT DESCRIPTION

Describes how to create and use a Windows PowerShell profile.

LONG DESCRIPTION

You can create a Windows PowerShell profile to customize your environment and to add session-specific elements to every Windows PowerShell session that you start.

A Windows PowerShell profile is a script that runs when Windows PowerShell starts. You can use the profile as a logon script to customize the environment. You can add commands, aliases, functions, variables, snap-ins, modules, and Windows PowerShell drives. You can also add other session-specific elements to your profile so they are available in every session without having to import or re-create them.

Windows PowerShell supports several profiles for users and host programs. However, it does not create the profiles for you. This topic describes the profiles, and it describes how to create and maintain profiles on your computer.

It explains how to use the NoProfile parameter of the Windows PowerShell console (PowerShell.exe) to start Windows PowerShell without any profiles. And, it explains the effect of the Windows PowerShell execution policy on profiles.

THE PROFILE FILES

Windows PowerShell supports several profile files. Also, Windows PowerShell host programs can support their own host-specific profiles.

For example, the Windows PowerShell console supports the following basic profile files. The profiles are listed in precedence order. The first profile has the highest precedence.

Description	Path
Current User, Current Host	$Home\[My]Documents\WindowsPowerShell\Profile.ps1
Current User, All Hosts	$Home\[My]Documents\Profile.ps1
All Users, Current Host	$PsHome\Microsoft.PowerShell_profile.ps1
All Users, All Hosts	$PsHome\Profile.ps1

The profile paths include the following variables:

- The $PsHome variable, which stores the installation directory for Windows PowerShell.

- The $Home variable, which stores the current user's home directory.

In addition, other programs that host Windows PowerShell can support their own profiles. For example, Windows PowerShell Integrated Scripting Environment (ISE) supports the following host-specific profiles.

```
Description            Path
-----------            -----
Current user, Current Host $Home\[My
]Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1
All users, Current Host    $PsHome\Microsoft.PowerShellISE_profile.ps1
```

In Windows PowerShell Help, the "CurrentUser, Current Host" profile is the profile most often referred to as "your Windows PowerShell profile".

THE $PROFILE VARIABLE

The $Profile automatic variable stores the paths to the Windows PowerShell profiles that are available in the current session.

To view a profile path, display the value of the $Profile variable. You can also use the $Profile variable in a command to represent a path.

The $Profile variable stores the path to the "Current User, Current Host" profile. The other profiles are saved in note properties of the $profile variable.

For example, the $Profile variable has the following values in the Windows PowerShell console.

```
Name                        Description
-----------                 -----------
$Profile                    Current User,Current Host
$Profile.CurrentUserCurrentHost    Current User,Current Host
$Profile.CurrentUserAllHosts       Current User,All Hosts
$Profile.AllUsersCurrentHost       All Users, Current Host
$Profile.AllUsersAllHosts          All Users, All Hosts
```

Because the values of the $Profile variable change for each user and in each host application, ensure that you display the values of the profile variables in each Windows PowerShell host application that you use.

To see the current values of the $Profile variable, type:

```
$profile | get-member -type noteproperty
```

You can use the $Profile variable in many commands. For example, the following command opens the "Current User, Current Host" profile in Notepad:

```
notepad $profile
```

The following command determines whether an "All Users, All Hosts" profile has been created on the local computer:

```
test-path $profile.AllUsersAllHosts
```

HOW TO CREATE A PROFILE

To create a Windows PowerShell profile, use the following command format:

```
if (!(test-path <profile-name>))
  {new-item -type file -path <profile-name> -force}
```

For example, to create a profile for the current user in the current Windows PowerShell host application, use the following command:

```
if (!(test-path $profile))
  {new-item -type file -path $profile -force}
```

In this command, the If statement prevents you from overwriting an existing profile. Replace the value of the <profile-path> placeholder with the path to the profile file that you want to create.

Note: To create "All Users" profiles in Windows Vista and later versions of Windows, start Windows PowerShell with the "Run as administrator" option.

HOW TO EDIT A PROFILE

You can open any Windows PowerShell profile in a text editor, such as Notepad.

To open the profile of the current user in the current Windows PowerShell host application in Notepad, type:

```
notepad $profile
```

To open other profiles, specify the profile name. For example, to open the profile for all the users of all the host applications, type:

```
notepad $profile.AllUsersAllHosts
```

To apply the changes, save the profile file, and then restart Windows PowerShell.

HOW TO CHOOSE A PROFILE

If you use multiple host applications, put the items that you use in all the host applications into your $Profile.CurrentUserAllHosts profile. Put items that are specific to a host application, such as a command that sets the background color for a host application, in a profile that is specific to that host application.

If you are an administrator who is customizing Windows PowerShell for many users, follow these guidelines:

-- Store the common items in the $profile.AllUsersAllHosts profile.

-- Store items that are specific to a host application in $profile.AllUsersCurrentHost profiles that are specific to the host application.

-- Store items for particular users in the user-specific profiles.

Be sure to check the host application documentation for any special implementation of Windows PowerShell profiles.

HOW TO USE A PROFILE

Many of the items that you create in Windows PowerShell and most commands that you run affect only the current session. When you end the session, the items are deleted.

The session-specific commands and items include variables, preference variables, aliases, functions, commands (except for Set-ExecutionPolicy), and Windows PowerShell snap-ins that you add to the session.

To save these items and make them available in all future sessions, add them to a Windows PowerShell profile.

Another common use for profiles is to save frequently-used functions, aliases, and variables. When you save the items in a profile, you can use them in any applicable session without re-creating them.

HOW TO START A PROFILE

When you open the profile file, it is blank. However, you can fill it with the variables, aliases, and commands that you use frequently.

Here are a few suggestions to get you started.

-- Add commands that make it easy to open your profile. This is especially useful if you use a profile other than the "Current User, Current Host" profile. For example, add the following command:

```
function pro {notepad $profile.CurrentUserAllHosts}
```

-- Add a function that opens Windows PowerShell Help in a compiled HTML Help file (.chm).

```
function Get-CHM
 {
   (invoke-item $env:windir\help\mui\0409\WindowsPowerShellHelp.chm)
 }
```

This function opens the English version of the .chm file. However, you can replace the language code (0409) to open other versions of the .chm file.

-- Add a function that lists the aliases for any cmdlet.

```
function Get-CmdletAlias ($cmdletname)
{
  get-alias | Where {$_.definition -like "*$cmdletname*"} | ft Definition, Name -auto
}
```

-- Add an Add-PsSnapin command to add any Windows PowerShell snap-ins that you use.

-- Customize your console.

```
function Color-Console
{
$host.ui.rawui.backgroundcolor = "white"
$host.ui.rawui.foregroundcolor = "black"
    $hosttime = (dir $pshome\PowerShell.exe).creationtime
    $Host.UI.RawUI.WindowTitle = "Windows PowerShell $hostversion ($hosttime)"
    clear-host
}
Color-console
```

-- Add a customized Windows PowerShell prompt that includes the computer name and the current path.

```
function prompt
{
  $env:computername + "\" + (get-location) + "> "
}
```

For more information about the Windows PowerShell prompt, see about_Prompts.

THE NOPROFILE PARAMETER

To start Windows PowerShell without profiles, use the NoProfile parameter of PowerShell.exe, the program that starts Windows PowerShell.

To begin, open a program that can start Windows PowerShell, such as Cmd.exe or Windows PowerShell itself. You can also use the Run dialog box in Windows.

Type:

```
PowerShell -noprofile
```

For a complete list of the parameters of PowerShell.exe, type:

```
PowerShell -?
```

PROFILES AND EXECUTION POLICY

The Windows PowerShell execution policy determines, in part, whether you can run scripts and load configuration files, including the profiles. The Restricted execution policy is the default. It prevents all scripts from running, including the profiles. If you use the Restricted policy, the profile does not run, and its contents are not applied.

A Set-ExecutionPolicy command sets and changes your execution policy. It is one of the few commands that applies in all Windows PowerShell sessions because the value is saved in the registry. You do not have to set it when you open the console, and you do not have to store a Set-ExecutionPolicy command in your profile.

PROFILES AND REMOTE SESSIONS

Windows PowerShell profiles are not run automatically in remote sessions, so the commands that the profileS add are not present in the remote 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.

```
invoke-command -session $s {invoke-command
"$home\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1"}
```

After running this command, the commands that the profile adds to the session are available in $s.

SEE ALSO

about_Automatic_Variables
about_Functions
about_Prompts
about_Execution_Policies
about_Signing
about_Remote
Set-ExecutionPolicy

TOPIC

about_Prompts

SHORT DESCRIPTION

Describes the Prompt function and demonstrates how to create a custom Prompt function.

LONG DESCRIPTION

The Windows PowerShell command prompt indicates that Windows PowerShell is ready to run a command:

```
PS C:\>
```

The Windows PowerShell prompt is determined by the built-in Prompt function. You can customize the prompt by creating your own Prompt function and saving it in your Windows PowerShell profile.

ABOUT THE PROMPT FUNCTION

The Prompt function determines the appearance of the Windows PowerShell prompt. Windows PowerShell comes with a built-in Prompt function, but you can override it by defining your own Prompt function.

The Prompt function has the following syntax:

```
function Prompt { <function-body> }
```

The Prompt function must return an object. As a best practice, return a string or an object that is formatted as a string. The maximum recommended length is 80 characters.

For example, the following prompt function returns a "Hello, World"

string followed by a caret (>).

```
PS C:\> function prompt {"Hello, World > "}
Hello, World >
```

GETTING THE PROMPT FUNCTION

To get the Prompt function, use the Get-Command cmdlet or use the Get-Item cmdlet in the Function drive.

Functions are commands. So, you can use the Get-Command cmdlet to get functions, including the Prompt function.

For example:

```
PS C:\>Get-Command Prompt

CommandType     Name                                ModuleName
-----------     ----                                ----------
Function        prompt
```

To get the script that sets the value of the prompt, use the dot method to get the ScriptBlock property of the Prompt function.

For example:

```
PS C:\>(Get-Command Prompt).ScriptBlock

"PS $($executionContext.SessionState.Path.CurrentLocation)$('>' * ($nestedPromptLevel + 1)) "
# .Link
# http://go.microsoft.com/fwlink/?LinkID=225750
# .ExternalHelp System.Management.Automation.dll-help.xml
```

Like all functions, the Prompt function is stored in the Function: drive. To display the script that creates the current Prompt function, type:

```
(Get-Item function:prompt).ScriptBlock
```

THE DEFAULT PROMPT

The default prompt appears only when the Prompt function generates an error or does not return an object.

The default Windows PowerShell prompt is:

```
PS>
```

For example, the following command sets the Prompt function to $null, which is invalid. As a result, the default prompt appears.

```
PS C:\> function prompt {$null}
PS>
```

Because Windows PowerShell comes with a built-in prompt, you usually do not see the default prompt.

BUILT-IN PROMPT

Windows PowerShell includes a built-in prompt function.

In Windows PowerShell 3.0, the built-in prompt function is:

```
function prompt
{
  "PS $($executionContext.SessionState.Path.CurrentLocation)$('>' * ($nestedPromptLevel + 1)) "
}
```

This simplified prompt starts with "PS" followed by the current location, and one ">" for each nested prompt level.

In Windows PowerShell 2.0, the built-in prompt function is:

```
function prompt
{
  $(if (test-path variable:/PSDebugContext) { '[DBG]: ' }
  else { '' }) + 'PS ' + $(Get-Location) `
  + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
}
```

The function uses the Test-Path cmdlet to determine whether the $PSDebugContext automatic variable is populated. If $PSDebugContext is populated, you are in debugging mode, and "[DBG]" is added to the prompt, as follows:

```
[DBG] PS C:\ps-test>
```

If $PSDebugContext is not populated, the function adds "PS" to the prompt. And, the function uses the Get-Location cmdlet to get the current file system directory location. Then, it adds a right angle bracket (>).

For example:

```
PS C:\ps-test>
```

If you are in a nested prompt, the function adds two angle brackets (>>) to the prompt. (You are in a nested prompt if the value of the $NestedPromptLevel automatic variable is greater than 1.)

For example, when you are debugging in a nested prompt, the prompt resembles the following prompt:

```
[DBG] PS C:\ps-test>>>
```

CHANGES TO THE PROMPT

The Enter-PSSession cmdlet prepends the name of the remote computer to the current Prompt function. When you use the Enter-PSSession cmdlet to start a session with a remote computer, the command prompt changes to include the name of the remote computer. For example:

```
PS Hello, World> Enter-PSSession Server01
[Server01]: PS Hello, World>
```

Other Windows PowerShell host applications and alternate shells might have their own custom command prompts.

For more information about the $PSDebugContext and $NestedPromptLevel automatic variables, see about_Automatic_Variables.

HOW TO CUSTOMIZE THE PROMPT

To customize the prompt, write a new Prompt function. The function is not protected, so you can overwrite it.

To write a prompt function, type the following:

```
function prompt { }
```

Then, between the braces, enter the commands or the string that creates your prompt.

For example, the following prompt includes your computer name:

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

On the Server01 computer, the prompt resembles the following prompt:

```
PS [Server01] >
```

The following prompt function includes the current date and time:

```
function prompt {"$(get-date)> "}
```

The prompt resembles the following prompt:

```
03/15/2012 17:49:47>
```

You can also change the default Prompt function:

For example, the following modified Prompt function adds "[ADMIN]:" to the built-in Windows PowerShell prompt when Windows PowerShell is opened by using the "Run as administrator" option:

```
function prompt
{
   $identity = [Security.Principal.WindowsIdentity]::GetCurrent()
   $principal = [Security.Principal.WindowsPrincipal] $identity

   $(if (test-path variable:/PSDebugContext) { '[DBG]: ' }

   elseif($principal.IsInRole([Security.Principal.WindowsBuiltInRole] "Administrator"))
   { "[ADMIN]: " }

   else { '' }) + 'PS ' + $(Get-Location) + $(if ($nestedpromptlevel -ge 1) { '>>' }) + '> '
}
```

When you start Windows PowerShell by using the "Run as administrator" option, a prompt that resembles the following prompt appears:

```
[ADMIN]: PS C:\ps-test>
```

The following Prompt function displays the history ID of the next command. To view the command history, use the Get-History cmdlet.

```
function prompt
{
  # The at sign creates an array in case only one history item exists.
  $history = @(get-history)
  if($history.Count -gt 0)
  {
    $lastItem = $history[$history.Count - 1]
    $lastId = $lastItem.Id
  }
```

```
    $nextCommand = $lastId + 1
    $currentDirectory = get-location
    "PS: $nextCommand $currentDirectory >"
  }
```

The following prompt uses the Write-Host and Get-Random cmdlets to create a prompt that changes color randomly. Because Write-Host writes to the current host application but does not return an object, this function includes a Return statement. Without it, Windows PowerShell uses the default prompt, "PS>".

```
function prompt
{
  $color = Get-Random -Min 1 -Max 16
  Write-Host ("PS " + $(Get-Location) +">") -NoNewLine -ForegroundColor $Color
  return " "
}
```

SAVING THE PROMPT FUNCTION

Like any function, the Prompt function exists only in the current session. To save the Prompt function for future sessions, add it to your Windows PowerShell profiles. For more information about profiles, see about_Profiles.

SEE ALSO

Get-Location
Enter-PSSession
Get-History
Get-Random
Write-Host
about_Profiles
about_Functions
about_Scopes
about_Debuggers
about_Automatic_Variables

TOPIC

about_Properties

SHORT DESCRIPTION

Describes how to use object properties in Windows PowerShell.

LONG DESCRIPTION

Windows PowerShell uses structured collections of information called objects to represent the items in data stores or the state of the computer. Typically, you work with object that are part of the Microsoft .NET Framework, but you can also create custom objects in Windows PowerShell.

The association between an item and its object is very close. When you change an object, you usually change the item that it represents. For example, when you get a file in Windows PowerShell, you do not get the actual file. Instead, you get a FileInfo object that represents the file. When you change the FileInfo object, the file changes too.

Most objects have properties. Properties are the data that is associated with an object. Different types of object have different properties. For example, a FileInfo object, which represents a file, has an IsReadOnly property that contains $True if the file the read-only attribute and $False if it does not. A DirectoryInfo object, which represents a file system directory, has a Parent property that contains the path to the parent directory.

OBJECT PROPERTIES

To get the properties of an object, use the Get-Member cmdlet. For example, to get the properties of a FileInfo object, use the Get-ChildItem cmdlet to get the FileInfo object that represents a file. Then, use a pipeline operator (|) to send the FileInfo object to Get-Member. The following command gets the PowerShell.exe file and sends it to Get-Member. The $Pshome automatic variable contains the path of the Windows PowerShell installation directory.

```
Get-ChildItem $pshome\PowerShell.exe | Get-Member
```

The output of the command lists the members of the FileInfo object. Members include both properties and methods. When you work in Windows PowerShell, you have access to all the members of the objects.

To get only the properties of an object and not the methods, use the MemberType parameter of the Get-Member cmdlet with a value of "property", as shown in the following example.

```
Get-ChildItem $pshome\PowerShell.exe | Get-Member -MemberType property

   TypeName: System.IO.FileInfo

Name           MemberType Definition
----           ---------- ----------
Attributes       Property   System.IO.FileAttributes Attributes {get;set;}
CreationTime     Property   System.DateTime CreationTime {get;set;}
CreationTimeUtc  Property   System.DateTime CreationTimeUtc {get;set;}
Directory        Property   System.IO.DirectoryInfo Directory {get;}
DirectoryName    Property   System.String DirectoryName {get;}
Exists           Property   System.Boolean Exists {get;}
Extension        Property   System.String Extension {get;}
FullName         Property   System.String FullName {get;}
IsReadOnly       Property   System.Boolean IsReadOnly {get;set;}
LastAccessTime   Property   System.DateTime LastAccessTime {get;set;}
LastAccessTimeUtc Property   System.DateTime LastAccessTimeUtc {get;set;}
LastWriteTime    Property   System.DateTime LastWriteTime {get;set;}
LastWriteTimeUtc  Property   System.DateTime LastWriteTimeUtc {get;set;}
Length           Property   System.Int64 Length {get;}
Name             Property   System.String Name {get;}
```

After you find the properties, you can use them in your Windows PowerShell commands.

PROPERTY VALUES

Although every object of a specific type has the same properties, the values of those properties describe the particular object. For example, every FileInfo object has a CreationTime property, but the value of that property differs for each file.

The most common way to get the values of the properties of an object is to use the dot method. Type a reference to the object, such as a variable that contains the object, or a command that gets the object. Then, type a dot (.) followed by the property name.

For example, the following command displays the value of the CreationTime property of the PowerShell.exe file. The Get-ChildItem command returns a FileInfo object that represents the PowerShell.exe file. The command is enclosed in parentheses to make sure that it is executed before any

properties are accessed. The Get-ChildItem command is followed by a dot and the name of the CreationTime property, as follows:

```
C:\PS> (Get-ChildItem $pshome\PowerShell.exe).creationtime
Tuesday, March 18, 2008 12:07:52 AM
```

You can also save an object in a variable and then get its properties by using the dot method, as shown in the following example:

```
C:\PS> $a = Get-ChildItem $pshome\PowerShell.exe
C:\PS> $a.CreationTime
Tuesday, March 18, 2008 12:07:52 AM
```

You can also use the Select-Object and Format-List cmdlets to display the property values of an object. Select-Object and Format-List each have a Property parameter. You can use the Property parameter to specify one or more properties and their values. Or, you can use the wildcard character (*) to represent all the properties.

For example, the following command displays the values of all the properties of the PowerShell.exe file.

```
C:\PS> Get-ChildItem $pshome\PowerShell.exe | Format-List -property *

PSPath          : Microsoft.PowerShell.Core\FileSystem::C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell.exe
PSParentPath    : Microsoft.PowerShell.Core\FileSystem::C:\Windows\system32\WindowsPowerShell\v1.0
PSChildName     : PowerShell.exe
PSDrive         : C
PSProvider      : Microsoft.PowerShell.Core\FileSystem
PSIsContainer   : False
VersionInfo     : File:           C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell.exe
                  InternalName:    POWERSHELL
                  OriginalFilename: PowerShell.EXE.MUI
                  File Version:    6.1.6570.1 (fbl_srv_PowerShell(nigels).070711-0102)
                  FileDescription:  PowerShell.EXE
                  Product:         Microsoft® Windows® Operating System
                  ProductVersion:   6.1.6570.1
                  Debug:           False
                  Patched:         False
                  PreRelease:      False
                  PrivateBuild:    True
```

```
                SpecialBuild:    False
                Language:        English (United States)

    BaseName        : PowerShell
    Mode            : -a---
    Name            : PowerShell.exe
    Length          : 160256
    DirectoryName   : C:\Windows\system32\WindowsPowerShell\v1.0
    Directory       : C:\Windows\system32\WindowsPowerShell\v1.0
    IsReadOnly      : False
    Exists          : True
    FullName        : C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell.exe
    Extension       : .exe
    CreationTime    : 3/18/2008 12:07:52 AM
    CreationTimeUtc : 3/18/2008 7:07:52 AM
    LastAccessTime  : 3/19/2008 8:13:58 AM
    LastAccessTimeUtc : 3/19/2008 3:13:58 PM
    LastWriteTime   : 3/18/2008 12:07:52 AM
    LastWriteTimeUtc : 3/18/2008 7:07:52 AM
    Attributes      : Archive
```

STATIC PROPERTIES

You can use the static properties of .NET classes in Windows PowerShell. Static properties are properties of class, unlike standard properties, which are properties of an object.

To get the static properties of an class, use the Static parameter of the Get-Member cmdlet.

For example, the following command gets the static properties of the System.DateTime class.

```
Get-Date | Get-Member -MemberType Property -Static

   TypeName: System.DateTime

Name     MemberType Definition
----     ---------- ----------
MaxValue Property   static datetime MaxValue {get;}
MinValue Property   static datetime MinValue {get;}
Now      Property   datetime Now {get;}
Today    Property   datetime Today {get;}
UtcNow   Property   datetime UtcNow {get;}
```

To get the value of a static property, use the following

syntax.

    [<ClassName>]::<Property>

For example, the following command gets the value of the
UtcNow static property of the System.DateTime class.

    [System.DateTime]::UtcNow

PROPERTIES OF SCALAR OBJECTS AND COLLECTIONS

The properties of one ("scalar") object of a particular type
are often different from the properties of a collection of objects
of the same type.

For example, every service has as DisplayName property, but
a collection of services does not have a DisplayName property.
Similarly, all collections have a Count property that tells how
many objects are in the collection, but individual objects do not
have a Count property.

Beginning in Windows PowerShell 3.0, Windows PowerShell tries
to prevent scripting errors that result from the differing
properties of scalar objects and collections.

-- If you submit a collection, but request a property
   that exists only on single ("scalar") objects, Windows
   PowerShell returns the value of that property for every object
   in the collection.

-- If you request the Count or Length property of zero objects
   or of one object, Windows PowerShell returns the correct value.

If the property exists on the individual objects and on the
collection, Windows PowerShell does not alter the result.

This feature also works on methods of scalar objects and
collections. For more information, see about_Methods.

EXAMPLES

For example, each service has a DisplayName property. The
following command gets the value of the DisplayName property
of the Audiosrv service.

```
PS C:\>(Get-Service Audiosrv).DisplayName
Windows Audio
```

However, a collection or array of services does not have a DisplayName. The following command tries to get the DisplayName property of all services in Windows PowerShell 2.0.

```
PS C:\>(Get-Service).DisplayName
PS C:\>
```

Beginning in Windows PowerShell 3.0, the same command returns the value of the DisplayName property of every service that Get-Service returns.

```
PS C:\>(Get-Service).DisplayName
Application Experience
Application Layer Gateway Service
Windows All-User Install Agent
Application Identity
Application Information
...
```

Conversely, a collection of two or more services has a Count property, which contains the number of objects in the collection.

```
PS C:\>(Get-Service).Count
176
```

Individual services do not have a Count or Length property, as shown in this command in Windows PowerShell 2.0.

```
PS C:\>(Get-Service Audiosrv).Count
PS C:\>
```

Beginning in Windows PowerShell 3.0, the command returns the correct Count value.

```
PS C:\>(Get-Service Audiosrv).Count
1
```

SEE ALSO

about_Methods
about_Objects
Get-Member
Select-Object
Format-List

TOPIC

about_Providers

SHORT DESCRIPTION

Describes how Windows PowerShell providers provide 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.

LONG DESCRIPTION

Windows PowerShell providers are Microsoft .NET Framework-based programs that make the data in a specialized data store available in Windows PowerShell so that you can view and manage it.

The data that a provider exposes appears in a drive, and you access the data in a path like you would on a hard disk drive. You can use any of the built-in cmdlets that the provider supports to manage the data in the provider drive. And, you can use custom cmdlets that are designed especially for the data.

The providers can also add dynamic parameters to the built-in cmdlets. These are parameters that are available only when you use the cmdlet with the provider data.

BUILT-IN PROVIDERS

Windows PowerShell includes a set of built-in providers that you can use to access the different types of data stores.

Provider Drive Data store

-------- ----- ----------
Alias Alias: Windows PowerShell aliases

Certificate Cert: x509 certificates for digital signatures

Environment Env: Windows environment variables

FileSystem * File system drives, directories, and files

Function Function: Windows PowerShell functions

Registry HKLM:, HKCU: Windows registry

Variable Variable: Windows PowerShell variables

WSMan WSMan: WS-Management configuration information

* The FileSystem drives vary on each system.

You can also create your own Windows PowerShell providers, and you can install providers that others develop. To list the providers that are available in your session, type:

get-psprovider

INSTALLING AND REMOVING PROVIDERS

Windows PowerShell providers are delivered to you in Windows PowerShell snap-ins, which are .NET Framework-based programs that are compiled into .dll files. The snap-ins can include providers and cmdlets.

Before you use the provider features, you have to install the snap-in and then add it to your Windows PowerShell session. For more information, see about_PSSnapins.

You cannot uninstall a provider, although you can remove the Windows PowerShell snap-in for the provider from the current session. If you do, you will remove all the contents of the snap-in, including its cmdlets.

To remove a provider from the current session, use the Remove-PSSnapin cmdlet. This cmdlet does not uninstall the provider, but it makes the provider unavailable in the session.

You can also use the Remove-PSDrive cmdlet to remove any drive from the current session. This data on the drive is not affected, but the drive is no longer available in that session.

VIEWING PROVIDERS

To view the Windows PowerShell providers on your computer, type:

```
get-psprovider
```

The output lists the built-in providers and the providers that you added to the session.

THE PROVIDER CMDLETS

The following cmdlets are designed to work with the data exposed by any provider. You can use the same cmdlets in the same way to manage the different types of data that providers expose. After you learn to manage the data of one provider, you can use the same procedures with the data from any provider.

For example, the New-Item cmdlet creates a new item. In the C: drive that is supported by the FileSystem provider, you can use New-Item to create a new file or folder. In the drives that are supported by the Registry provider, you can use New-Item to create a new registry key. In the Alias: drive, you can use New-Item to create a new alias.

For detailed information about any of the following cmdlets, type:

```
get-help <cmdlet-name> -detailed
```

CHILDITEM CMDLETS

Get-ChildItem

CONTENT CMDLETS

Add-Content
Clear-Content
Get-Content
Set-Content

ITEM CMDLETS

Clear-Item
Copy-Item
Get-Item

Invoke-Item
Move-Item
New-Item
Remove-Item
Rename-Item
Set-Item

ITEMPROPERTY CMDLETS

Clear-ItemProperty
Copy-ItemProperty
Get-ItemProperty
Move-ItemProperty
New-ItemProperty
Remove-ItemProperty
Rename-ItemProperty
Set-ItemProperty

LOCATION CMDLETS

Get-Location
Pop-Location
Push-Location
Set-Location

PATH CMDLETS

Join-Path
Convert-Path
Split-Path
Resolve-Path
Test-Path

PSDRIVE CMDLETS

Get-PSDrive
New-PSDrive
Remove-PSDrive

PSPROVIDER CMDLETS

Get-PSProvider

VIEWING PROVIDER DATA

The primary benefit of a provider is that it exposes its data in a familiar and consistent way. The model for data presentation is a file system drive.

To use data that the provider exposes, you view it, move through it, and change it as though it were data on a hard drive. Therefore, the most important information about a provider is the name of the drive

that it supports.

The drive is listed in the default display of the Get-PSProvider cmdlet, but you can get information about the provider drive by using the Get-PSDrive cmdlet. For example, to get all the properties of the Function: drive, type:

```
get-psdrive Function | format-list *
```

You can view and move through the data in a provider drive just as you would on a file system drive.

To view the contents of a provider drive, use the Get-Item or Get-ChildItem cmdlets. Type the drive name followed by a colon (:). For example, to view the contents of the Alias: drive, type:

```
get-item alias:
```

You can view and manage the data in any drive from another drive by including the drive name in the path. For example, to view the HKLM\Software registry key in the HKLM: drive from another drive, type:

```
get-childitem hklm:\software
```

To open the drive, use the Set-Location cmdlet. Remember the colon when you specify the drive path. For example, to change your location to the root directory of the Cert: drive, type:

```
set-location cert:
```

Then, to view the contents of the Cert: drive, type:

```
get-childitem
```

MOVING THROUGH HIERARCHICAL DATA

You can move through a provider drive just as you would a hard disk drive. If the data is arranged in a hierarchy of items within items, use a backslash (\) to indicate a child item. Use the following format:

```
drive:\location\child-location\...
```

For example, to change your location to the HKLM\Software registry key, type a Set-Location command, such as:

```
set-location hklm:\software
```

You can also use relative references to locations. A dot (.) represents the current location. For example, if you are in the HKLM:\Software\Microsoft registry key, and you want to list the registry subkeys in the HKLM:\Software\Microsoft\PowerShell key, type the following command:

```
get-childitem .\PowerShell
```

FINDING DYNAMIC PARAMETERS

Dynamic parameters are cmdlet parameters that are added to a cmdlet by a provider. These parameters are available only when the cmdlet is used with the provider that added them.

For example, the Cert: drive adds the CodeSigningCert parameter to the Get-Item and Get-ChildItem cmdlets. You can use this parameter only when you use Get-Item or Get-ChildItem in the Cert: drive.

For a list of the dynamic parameters that a provider supports, see the Help file for the provider. Type:

```
get-help <provider-name>
```

For example:

```
get-help certificate
```

LEARNING ABOUT PROVIDERS

Although all provider data appears in drives, and you use the same methods to move through them, the similarity stops there. The data stores that the provider exposes can be as varied as Active Directory locations and Microsoft Exchange Server mailboxes.

For information about individual Windows PowerShell providers, type:

```
get-help <ProviderName>
```

For example:

```
get-help registry
```

For a list of Help topics about the providers, type:

```
get-help * -category provider
```

SEE ALSO

about_Locations
about_Path_Syntax

TOPIC

about_PSConsoleHostReadLine

SHORT DESCRIPTION

Explains how to create a customize how Windows PowerShell reads input at the console prompt.

LONG DESCRIPTION

Starting in Windows PowerShell V3, you can write a function named PSConsoleHostReadLine that overrides the default way that console input is processed.

EXAMPLES

The following example launches Notepad and gets input from a text File that the user creates:

```
function PSConsoleHostReadLine
{
   $inputFile = Join-Path $env:TEMP PSConsoleHostReadLine
```

```
        Set-Content $inputFile "PS > "

        ## Notepad opens. Enter your command in it, save the file,
        ## and then exit.
        notepad $inputFile | Out-Null
        $userInput = Get-Content $inputFile
        $resultingCommand = $userInput.Replace("PS >", "")
        $resultingCommand
    }
```

REMARKS

By default, Windows PowerShell reads input from the console in what is known as “Cooked Mode”—in which the Windows console subsystem handles all the keypresses, F7 menus, and other input. When you press Enter or Tab, Windows PowerShell gets the text that you have typed up to that point. There is no way for it to know that you pressed Ctrl-R, Ctrl-A, Ctrl-E, or any other keys before pressing Enter or Tab. In Windows PowerShell version 3, the PSConsoleHostReadLine function solves this issue. When you define a function named PSConsoleHostReadline in the Windows PowerShell console host, Windows PowerShell calls that function instead of the “Cooked Mode” input mechanism.

SEE ALSO

about_Prompts

TOPIC

about_PSReadline

SHORT DESCRIPTION

PSReadline provides an improved command line editing experience in the Windows PowerShell console.

LONG DESCRIPTION

PSReadline, introduced in Windows PowerShell 5.0, provides a powerful
command line editing experience for the Windows PowerShell console.
It provides:

* Syntax coloring of the command line
* A visual indication of syntax errors
* A better multi-line experience (both editing and history)
* Customizable key bindings
* Cmd and Emacs modes
* Many configuration options
* Bash style completion (optional in Cmd mode, default in Emacs mode)
* Emacs yank/kill ring
* Windows PowerShell token based "word" movement and kill

The following functions are available in the class [PSConsoleUtilities.PSConsoleReadLine]:

Cursor movement

EndOfLine (Cmd: <End> Emacs: <End> or <Ctrl+E>)

If the input has multiple lines, move to the end of the current line,
or if already at the end of the line, move to the end of the input.
If the input has a single line, move to the end of the input.

BeginningOfLine (Cmd: <Home> Emacs: <Home> or <Ctrl+A>)

If the input has multiple lines, move to the start of the current line,
or if already at the start of the line, move to the start of the input.
If the input has a single line, move to the start of the input.

NextLine (Cmd: unbound Emacs: unbound)

Move the cursor to the next line if the input has multiple lines.

PreviousLine (Cmd: unbound Emacs: unbound)

Move the cursor to the previous line if the input has multiple lines.

ForwardChar (Cmd: <RightArrow> Emacs: <RightArrow> or <Ctrl+F>)

Move the cursor one character to the right. This might move the cursor to the next
line of multi-line input.

BackwardChar (Cmd: <LeftArrow> Emacs: <LeftArrow> or <Ctrl+B>)

Move the cursor one character to the left. This might move the cursor to the previous
line of multi-line input.

ForwardWord (Cmd: unbound Emacs: <Alt+F>)

Move the cursor forward to the end of the current word, or if between words, to the end of the next word. You can set word delimiter characters with:

Set-PSReadlineOption -WordDelimiters <string of delimiter characters>

NextWord (Cmd: <Ctrl+RightArrow> Emacs: unbound)

Move the cursor forward to the start of the next word. You can set word delimiter characters with:

Set-PSReadlineOption -WordDelimiters <string of delimiter characters>

BackwardWord (Cmd: <Ctrl+LeftArrow> Emacs: <Alt+B>)

Move the cursor back to the start of the current word, or if between words, the start of the previous word. You can set word delimiter characters with:

Set-PSReadlineOption -WordDelimiters <string of delimiter characters>

ShellForwardWord (Cmd: unbound Emacs: unbound)

Like ForwardWord except word boundaries are defined by Windows PowerShell token boundaries.

ShellNextWord (Cmd: unbound Emacs: unbound)

Like NextWord except word boundaries are defined by Windows PowerShell token boundaries.

ShellBackwardWord (Cmd: unbound Emacs: unbound)

Like BackwardWord except word boundaries are defined by Windows PowerShell token boundaries.

GotoBrace (Cmd: <Ctrl+}> Emacs: unbound)

Go to the matching parenthesis, curly brace, or square bracket.

AddLine (Cmd: <Shift-Enter> Emacs: <Shift-Enter>)

The continuation prompt is displayed on the next line and PSReadline waits for keys to edit the current input. This is useful to enter multi-line input as a single command even when a single line is complete input by itself.

Basic editing

CancelLine (Cmd: unbound Emacs: unbound)

Cancel all editing to the line, leave the line of input on the screen but return from PSReadline without executing the input.

RevertLine (Cmd: <ESC> Emacs: <Alt+R>)

Reverts all of the input since the last input was accepted and run. This is equivalent to using the Undo command until there is nothing left to undo.

BackwardDeleteChar (Cmd: <Backspace> Emacs: <Backspace> or <Ctrl+H>)

Delete the character before the cursor.

DeleteChar (Cmd: <Delete> Emacs: <Delete>)

Delete the character under the cursor.

DeleteCharOrExit (Cmd: unbound Emacs: <Ctrl+D>)

Like DeleteChar, unless the line is empty, in which case exit the process.

AcceptLine (Cmd: <Enter> Emacs: <Enter> or <Ctrl+M>)

Attempt to execute the current input. If the current input is incomplete (for example, there is a missing closing parenthesis, bracket, or quote), then the continuation prompt is displayed on the next line, and PSReadline waits for keys to edit the current input.

AcceptAndGetNext (Cmd: unbound Emacs: <Ctrl+O>)

Like AcceptLine, but after the line completes, start editing the next line from history.

ValidateAndAcceptLine (Cmd: unbound Emacs: unbound)

Like AcceptLine but performs additional validation including:

* Checks for additional parse errors
* Validates that command names are all found
* If you are running Windows PowerShell 4.0 or newer, validates the parameters and arguments

If there are any errors, the error message is displayed and not accepted nor added to the history unless you either correct the command line or execute AcceptLine or

ValidateAndAcceptLine again while the error message is displayed.

BackwardDeleteLine (Cmd: <Ctrl+Home> Emacs: unbound)

Delete the text from the start of the input to the cursor.

ForwardDeleteLine (Cmd: <Ctrl+End> Emacs: unbound)

Delete the text from the cursor to the end of the input.

SelectBackwardChar (Cmd: <Shift+LeftArrow> Emacs: <Shift+LeftArrow>)

Adjust the current selection to include the previous character.

SelectForwardChar (Cmd: <Shift+RightArrow> Emacs: <Shift+RightArrow>)

Adjust the current selection to include the next character.

SelectBackwardWord (Cmd: <Shift+Ctrl+LeftArrow> Emacs: <Alt+Shift+B>)

Adjust the current selection to include the previous word.

SelectForwardWord (Cmd: unbound Emacs: <Alt+Shift+F>)

Adjust the current selection to include the next word using ForwardWord.

SelectNextWord (Cmd: <Shift+Ctrl+RightArrow> Emacs: unbound)

Adjust the current selection to include the next word using NextWord.

SelectShellForwardWord (Cmd: unbound Emacs: unbound)

Adjust the current selection to include the next word using ShellForwardWord.

SelectShellNextWord (Cmd: unbound Emacs: unbound)

Adjust the current selection to include the next word using ShellNextWord.

SelectShellBackwardWord (Cmd: unbound Emacs: unbound)

Adjust the current selection to include the previous word using ShellBackwardWord.

SelectBackwardsLine (Cmd: <Shift+Home> Emacs: <Shift+Home>)

Adjust the current selection to include from the cursor to the start of the line.

SelectLine (Cmd: <Shift+End> Emacs: <Shift+End>)

Adjust the current selection to include from the cursor to the end of the line.

SelectAll (Cmd: <Ctrl+A> Emacs: unbound)

Select the entire line. Moves the cursor to the end of the line.

SelfInsert (Cmd: <a>, <b>, ... Emacs: <a>, <b>, ...)

Insert the key entered.

Redo (Cmd: <Ctrl+Y> Emacs: unbound)

Redo an insertion or deletion that was undone by Undo.

Undo (Cmd: <Ctrl+Z> Emacs: <Ctrl+_>)

Undo a previous insertion or deletion.

History

ClearHistory (Cmd: Alt+F7 Emacs: unbound)

Clears history in PSReadline. This does not affect Windows PowerShell history.

PreviousHistory (Cmd: <UpArrow> Emacs: <UpArrow> or <Ctrl+P>)

Replace the current input with the previous item from PSReadline history.

NextHistory (Cmd: <DownArrow> Emacs: <DownArrow> or <Ctrl+N>)

Replace the current input with the next item from PSReadline history.

ForwardSearchHistory (Cmd: <Ctrl+S> Emacs: <Ctrl+S>)

Search forward from the current history line interactively.

ReverseSearchHistory (Cmd: <Ctrl+R> Emacs: <Ctrl+R>)

Search backward from the current history line interactively.

HistorySearchBackward (Cmd: <F8> Emacs: unbound)

Replace the current input with the previous item from PSReadline history that matches the characters between the start and the input and the cursor.

HistorySearchForward (Cmd: <Shift+F8> Emacs: unbound)

Replace the current input with the next item from PSReadline history
that matches the characters between the start and the input and the cursor.

BeginningOfHistory (Cmd: unbound Emacs: <Alt+<>)

Replace the current input with the last item from PSReadline history.

EndOfHistory (Cmd: unbound Emacs: <Alt+>>)

Replace the current input with the last item in PSReadline history, which
is the possibly empty input that was entered before any history commands.

Tab Completion

TabCompleteNext (Cmd: <Tab> Emacs: unbound)

Attempt to complete the text surrounding the cursor with the next
available completion.

TabCompletePrevious (Cmd: <Shift-Tab> Emacs: unbound)

Attempt to complete the text surrounding the cursor with the next
previous completion.

Complete (Cmd: unbound Emacs: <Tab>)

Attempt to perform completion on the text surrounding the cursor.
If there are multiple possible completions, the longest unambiguous
prefix is used for completion. If you are trying to complete the longest
unambiguous completion, a list of possible completions is displayed.

MenuComplete (Cmd: <Ctrl+Space> Emacs: <Ctrl+Space>)

Attempt to perform completion on the text surrounding the cursor.
If there are multiple possible completions, a list of possible
completions is displayed, and you can select the correct completion
by using the arrow keys, or Tab/Shift+Tab. Escape and Ctrl+G cancel
the menu selection, and revert the line to the state before invoking
MenuComplete.

PossibleCompletions (Cmd: unbound Emacs: <Alt+Equals>)

Display the list of possible completions.

SetMark (Cmd: unbound Emacs: <Alt+Space>)

Mark the current location of the cursor for use in a subsequent editing command.

ExchangePointAndMark (Cmd: unbound Emacs: <Ctrl+X,Ctrl+X>)

The cursor is placed at the location of the mark and the mark is moved to the location of the cursor.

Kill/Yank

Kill and Yank operate on a clipboard in the PSReadline module. There is a ring buffer called the kill ring - killed text is added to the kill ring up and yank will copy text from the most recent kill. YankPop cycles through items in the kill ring. When the kill ring is full, new items replace the oldest items. A kill operation that is immediately preceded by another kill operation appends the previous kill, instead of adding a new item or replacing an item in the kill ring. This is how you can cut a part of a line, for example, with multiple KillWord operations, then yank them back elsewhere as a single yank.

KillLine (Cmd: unbound Emacs: <Ctrl+K>)

Clear the input from the cursor to the end of the line. The cleared text is placed in the kill ring.

BackwardKillLine (Cmd: unbound Emacs: <Ctrl+U> or <Ctrl+X,Backspace>)

Clear the input from the start of the input to the cursor. The cleared text is placed in the kill ring.

KillWord (Cmd: unbound Emacs: <Alt+D>)

Clear the input from the cursor to the end of the current word. If the cursor is between words, the input is cleared from the cursor to the end of the next word. The cleared text is placed in the kill ring.

BackwardKillWord (Cmd: unbound Emacs: <Alt+Backspace>)

Clear the input from the start of the current word to the cursor. If the cursor is between words, the input is cleared from the start of the previous word to the cursor. The cleared text is placed in the kill ring.

ShellKillWord (Cmd: unbound Emacs: unbound)

Like KillWord, except word boundaries are defined by Windows PowerShell token boundaries.

ShellBackwardKillWord (Cmd: unbound Emacs: unbound)

Like BackwardKillWord, except word boundaries are defined by Windows PowerShell token boundaries.

UnixWordRubout (Cmd: unbound Emacs: <Ctrl+W>)

Like BackwardKillWord, except word boundaries are defined by white space.

KillRegion (Cmd: unbound Emacs: unbound)

Kill the text between the cursor and the mark.

Copy (Cmd: <Ctrl+Shift+C> Emacs: unbound)

Copy selected region to the system clipboard. If no region is selected, copy the whole line.

CopyOrCancelLine (Cmd: <Ctrl+C> Emacs: <Ctrl+C>)

Either copy selected text to the clipboard, or if no text is selected, cancel editing the line with CancelLine.

Cut (Cmd: <Ctrl+X> Emacs: unbound)

Delete selected region placing deleted text in the system clipboard.

Yank (Cmd: unbound Emacs: <Ctrl+Y>)

Add the most-recently killed text to the input.

YankPop (Cmd: unbound Emacs: <Alt+Y>)

If the previous operation was Yank or YankPop, replace the previously-yanked text with the next killed text from the kill ring.

ClearKillRing (Cmd: unbound Emacs: unbound)

The contents of the kill ring are cleared.

Paste (Cmd: <Ctrl+V> Emacs: unbound)

This is similar to Yank, but uses the system clipboard instead of the kill ring.

YankLastArg (Cmd: <Alt+.> Emacs: <Alt+.>, <Alt+_>)

Insert the last argument from the previous command in history. Repeated operations replace the last inserted argument with the last argument from the previous command (so Alt+. Alt+. will insert the last argument of the second to last history line).

With an argument, the first time YankLastArg behaves like YankNthArg. A negative argument on subsequent YankLastArg calls changes the direction while going through history. For example, if you press Alt+. one too many times, you can type Alt+- Alt+. to reverse the direction.

Arguments are based on Windows PowerShell tokens.

YankNthArg (Cmd: unbound Emacs: <Alt+Ctrl+Y>)

Insert the first argument (not the command name) of the previous command in history.

With an argument, insert the nth argument where 0 is typically the command. Negative arguments start from the end.

Arguments are based on Windows PowerShell tokens.

Miscellaneous

Abort (Cmd: unbound Emacs: <Ctrl+G>)

Abort the current action; for example, stop interactive history search.
Does not cancel input like CancelLine.

CharacterSearch (Cmd: <F3> Emacs: <Ctrl+]>)

Read a key and search forwards for that character. With an argument, search forwards for the nth occurrence of that argument. With a negative argument, searches backwards.

CharacterSearchBackward (Cmd: <Shift+F3> Emacs: <Alt+Ctrl+]>)

Like CharacterSearch, but searches backwards. With a negative argument, searches forwards.

ClearScreen (Cmd: <Ctrl+L> Emacs: <Ctrl+L>)

Clears the screen and displays the current prompt and input at the top of the screen.

DigitArgument (Cmd: unbound Emacs: <Alt+[0..9]>,,<Alt+->)

Used to pass numeric arguments to functions like CharacterSearch or YankNthArg. Alt+- toggles the argument to be negative/non-negative. To enter 80 '*' characters, you could type Alt+8 Alt+0 *.

CaptureScreen (Cmd: unbound Emacs: unbound)

Copies selected lines to the clipboard in both text and RTF formats. Use up/down

arrow keys to the first line to select, then Shift+UpArrow/Shift+DownArrow to select multiple lines. After selecting, press Enter to copy the text. Escape/Ctrl+C/Ctrl+G cancel the operation, so nothing is copied to the clipboard.

InvokePrompt (Cmd: unbound Emacs: unbound)

Erases the current prompt and calls the prompt function to redisplay the prompt. Useful for custom key handlers that change state, such as changing the current directory.

WhatIsKey (Cmd: <Alt+?> Emacs: <Alt+?>)

Read a key or chord and display the key binding.

ShowKeyBindings (Cmd: <Ctrl+Alt+?> Emacs: <Ctrl+Alt+?>)

Show all of the currently-bound keys.

ScrollDisplayUp (Cmd: <PageUp> Emacs: <PageUp>)

Scroll the display up one screen.

ScrollDisplayUpLine (Cmd: <Ctrl+PageUp> Emacs: <Ctrl+PageUp>)

Scroll the display up one line.

ScrollDisplayDown (Cmd: <PageDown> Emacs: <PageDown>)

Scroll the display down one screen.

ScrollDisplayDownLine (Cmd: <Ctrl+PageDown> Emacs: <Ctrl+PageDown>)

Scroll the display down one line.

ScrollDisplayTop (Cmd: unbound Emacs: <Ctrl+Home>)

Scroll the display to the top.

ScrollDisplayToCursor (Cmd: unbound Emacs: <Ctrl+End>)

Scroll the display to the cursor.

Custom Key Bindings

PSReadline supports custom key bindings using the cmdlet Set-PSReadlineKeyHandler. Most custom key bindings call one of the above functions, for example:

```
Set-PSReadlineKeyHandler -Key UpArrow -Function HistorySearchBackward
```

You can bind a ScriptBlock to a key. The ScriptBlock can do virtually anything you want.
Some useful examples include:

* edit the command line
* opening a new window (e.g. help)
* change directories without changing the command line

The ScriptBlock is passed two arguments:

* $key - A [ConsoleKeyInfo] that is the key that triggered the custom binding. If you bind the same ScriptBlock to multiple keys and need to perform different actions depending on the key, you can check $key. Many custom bindings ignore this argument.
* $arg - An arbitrary argument. Most often, this would be an integer argument that the user passes from the key bindings DigitArgument. If your binding doesn't accept arguments, it's reasonable to ignore this argument.

Let's take a look at an example that adds a command line to history without executing it. This is useful when you realize you forgot to do something, but don't want to re-enter the command line you've already entered.

```
Set-PSReadlineKeyHandler -Key Alt+w `
                         -BriefDescription SaveInHistory `
                         -LongDescription "Save current line in history but do not execute" `
                         -ScriptBlock {
    param($key, $arg)   # The arguments are ignored in this example

    # We need the command line, GetBufferState gives us that (with the cursor position)
    $line = $null
    $cursor = $null
    [PSConsoleUtilities.PSConsoleReadLine]::GetBufferState([ref]$line, [ref]$cursor)

    # AddToHistory saves the line in history, but does not execute the line.
    [PSConsoleUtilities.PSConsoleReadLine]::AddToHistory($line)

    # RevertLine is like pressing Escape.
    [PSConsoleUtilities.PSConsoleReadLine]::RevertLine()
}
```

You can see many more examples in the file SamplePSReadlineProfile.ps1 which is installed in the PSReadline module folder.

Most key bindings will want to take advantage of some helper functions for editing the command line those APIs are documented in the next section.

Custom Key Binding Support APIs

The following functions are public in PSConsoleUtilities.PSConsoleReadline, but cannot be directly bound to a key. Most are useful in custom key bindings.

void AddToHistory(string command)

Add a command line to history without executing it.

void ClearKillRing()

Clear the kill ring. This is mostly used for testing.

void Delete(int start, int length)

Delete length characters from start. This operation supports undo and redo.

void Ding()

Perform the Ding action based on the user's preference.

void GetBufferState([ref] string input, [ref] int cursor)
void GetBufferState([ref] Ast ast, [ref] Token[] tokens, [ref] ParseError[] parseErrors, [ref] int cursor)

These two functions retrieve useful information about the current state of the input buffer. The first is more commonly used for simple cases. The second is used if your binding is doing something more advanced with the Ast.

IEnumerable[PSConsoleUtilities.KeyHandler] GetKeyHandlers(bool includeBound, bool includeUnbound)

This function is used by Get-PSReadlineKeyHandler and probably isn't useful in a custom key binding.

PSConsoleUtilities.PSConsoleReadlineOptions GetOptions()

This function is used by Get-PSReadlineOption and probably isn't too useful in a custom key binding.

void GetSelectionState([ref] int start, [ref] int length)

If there is no selection on the command line, -1 is returned in both start and length.
If there is a selection on the command line, the start and length of the selection are returned.

void Insert(char c)
void Insert(string s)

Insert a character or string at the cursor. This operation supports undo and redo.

string ReadLine(runspace remoteRunspace, System.Management.Automation.EngineIntrinsics engineIntrinsics)

This is the main entry point to PSReadline. It does not support recursion, so is not useful in a custom key binding.

void RemoveKeyHandler(string[] key)

This function is used by Remove-PSReadlineKeyHandler and probably isn't too useful in a custom key binding.

void Replace(int start, int length, string replacement)

Replace some of the input. This operation supports undo and redo.
This is preferred over Delete followed by Insert because it is treated as a single action for undo.

void SetCursorPosition(int cursor)

Move the cursor to the given offset. Cursor movement is not tracked for undo.

void SetOptions(PSConsoleUtilities.SetPSReadlineOption options)

This function is a helper method used by the cmdlet Set-PSReadlineOption, but might be useful to a custom key binding that wants to temporarily change a setting.

bool TryGetArgAsInt(System.Object arg, [ref] int numericArg, int defaultNumericArg)

This helper method is used for custom bindings that honor DigitArgument. A typical call looks like:

```
[int]$numericArg = 0
[PSConsoleUtilities.PSConsoleReadLine]::TryGetArgAsInt($arg, [ref]$numericArg, 1)
```

WINDOWS POWERSHELL COMPATIBILITY

PSReadline requires Windows PowerShell 3.0 or newer, and the console host. It does not work in Windows PowerShell ISE.

FEEDBACK

https://github.com/lzybkr/PSReadline

CONTRIBUTING TO PSREADLINE

Feel free to submit a pull request or submit feedback on the PSReadline GitHub page (https://github.com/lzybkr/PSReadLine).

SEE ALSO

PSReadline is heavily influenced by the GNU Readline library:
http://tiswww.case.edu/php/chet/readline/rltop.html

Get-PSReadlineKeyHandler
Get-PSReadlineOption
Remove-PSReadlineKeyHandler
Set-PSReadlineKeyHandler
Set-PSReadlineOption

TOPIC

about_PSSessions

SHORT DESCRIPTION

Describes Windows PowerShell sessions (PSSessions) and explains how to establish a persistent connection to a remote computer.

LONG DESCRIPTION

To run Windows PowerShell commands on a remote computer, you can use the ComputerName parameter of a cmdlet, or you can create a Windows PowerShell session (PSSession) and run commands in the PSSession.

When you create a PSSession, Windows PowerShell establishes a persistent connection to the remote computer. Use a PSSession to run a series of related commands on a remote computer. Commands that run in the same PSSession can share data, such as the values of variables, aliases, and functions.

You can also create a PSSession on the local computer and run commands in it. A local PSSession uses the Windows PowerShell remoting

infrastructure to create and maintain the PSSession.

Beginning in Windows PowerShell 3.0, PSSessions are independent of the sessions in which they are created. Active PSSessions are maintained on the remote computer (or the computer at the remote end or "server-side" of the connection). As a result, you can disconnect from the PSSession and reconnect to it at a later time from the same computer or from a different computer.

This topic explains how to create, use, get, and delete PSSessions. For more advanced information, see about_PSSession_Details.

Note: PSSessions use the Windows PowerShell remoting infrastructure. To use PSSessions, the local and remote computers must be configured for remoting. For more information, see about_Remote_Requirements.

In Windows Vista and later versions of Windows, to create a PSSession on a local computer, you must start Windows PowerShell with the "Run as administrator" option.

WHAT IS A SESSION?

A session is an environment in which Windows PowerShell runs.

Each time you start Windows PowerShell, a session is created for you, and you can run commands in the session. You can also add items to your session, such as modules and snap-ins, and you can create items, such as variables, functions, and aliases. These items exist only in the session and are deleted when the session ends.

You can also create user-managed sessions, known as "Windows PowerShell sessions" or "PSSessions," on the local computer or on a remote computer. Like the default session, you can run commands in a PSSession and add and create items.

However, unlike the session that starts automatically, you can control the PSSessions that you create. You can get, create, configure, and remove them, disconnect and reconnect to them, and run multiple commands in the same PSSession. The PSSession remains available until you delete it or it times out.

Typically, you create a PSSession to run a series of related commands on a remote computer. When you create a PSSession on a remote computer, Windows PowerShell establishes a persistent connection to the remote computer to support the session.

If you use the ComputerName parameter of the Invoke-Command or

Enter-PSSession cmdlet to run a remote command or to start an interactive session, Windows PowerShell creates a temporary session on the remote computer and closes the session as soon as the command is complete or as soon as the interactive session ends. You cannot control these temporary sessions, and you cannot use them for more than a single command or a single interactive session.

In Windows PowerShell, the "current session" is the session that you are working in. The "current session" can refer to any session, including a temporary session or a PSSession.

WHY USE A PSSESSION?

Use a PSSession when you need a persistent connection to a remote computer. With a PSSession, you can run a series of commands that share data, such as the value of variables, the contents of a function, or the definition of an alias.

You can run remote commands without creating a PSSession. Use the ComputerName parameter of remote-enabled cmdlets to run a single command or a series of unrelated commands on one or many computers.

When you use the ComputerName parameter of Invoke-Command or Enter-PSSession, Windows PowerShell establishes a temporary connection to the remote computer and then closes the connection as soon as the command is complete. Any data elements that you create are lost when the connection is closed.

Other cmdlets that have a ComputerName parameter, such as Get-Eventlog and Get-WmiObject, use different remoting technologies to gather data. None create a persistent connection like a PSSession.

HOW TO CREATE A PSSESSION

To create a PSSession, use the New-PSSession cmdlet. To create the PSSession on a remote computer, use the ComputerName parameter of the New-PSSession cmdlet.

For example, the following command creates a new PSSession on the Server01 computer.

```
New-PSSession -ComputerName Server01
```

When you submit the command, New-PSSession creates the PSSession and returns an object that represents the PSSession. You can save the object in a variable when you create the PSSession, or you can use a Get-PSSession command to get the PSSession at a later time.

For example, the following command creates a new PSSession on the Server01 computer and saves the resulting object in the $ps variable.

```
$ps = New-PSSession -ComputerName Server01
```

HOW TO CREATE PSSESSIONS ON MULTIPLE COMPUTERS

To create PSSessions on multiple computers, use the ComputerName parameter of the New-PSSession cmdlet. Type the names of the remote computers in a comma-separated list.

For example, to create PSSessions on the Server01, Server02, and Server03 computers, type:

```
New-PSSession -ComputerName Server01, Server02, Server03
```

New-PSSession creates one PSSession on each of the remote computers.

HOW TO GET PSSESSIONS

To get the PSSessions that were created in your current session, use the Get-PSSession cmdlet without the ComputerName parameter. Get-PSSession returns the same type of object that New-PSSession returns.

The following command gets all the PSSessions that were created in the current session.

```
Get-PSSession
```

The default display of the PSSessions shows their ID and a default display name. You can assign an alternate display name when you create the session.

```
Id  Name      ComputerName   State    ConfigurationName
--- ----      ------------   -----    ---------------------
1   Session1  Server01       Opened   Microsoft.PowerShell
2   Session2  Server02       Opened   Microsoft.PowerShell
3   Session3  Server03       Opened   Microsoft.PowerShell
```

You can also save the PSSessions in a variable. The following command gets the PSSessions and saves them in the $ps123 variable.

```
$ps123 = Get-PSSession
```

When using the PSSession cmdlets, you can refer to a PSSession by its ID, by its name, or by its instance ID (a GUID). The following command gets a PSSession by its ID and saves it in the $ps01 variable.

```
$ps01 = Get-PSSession -Id 1
```

Beginning in Windows PowerShell 3.0, PSSessions are maintained on the remote computer. To get PSSessions that you created on particular remote computers, use the ComputerName parameter of the Get-PSSession cmdlet. The following command gets the PSSessions that you created on the Server01 remote computer. This includes PSSessions created in the current session and in other sessions on the local computer or other computers.

```
Get-PSSession -ComputerName Server01
```

In Windows PowerShell 2.0, Get-PSSession gets only the PSSessions that were created in the current session. It does not get PSSessions that were created in other sessions or on other computers, even if the sessions are connected to and are running commands on the local computer.

HOW TO RUN COMMANDS IN A PSSESSION

To run a command in one or more PSSessions, use the Invoke-Command cmdlet. Use the Session parameter to specify the PSSessions and the ScriptBlock parameter to specify the command.

For example, to run a Get-ChildItem ("dir") command in each of the three PSSessions saved in the $ps123 variable, type:

```
Invoke-Command -Session $ps123 -ScriptBlock {Get-ChildItem}
```

HOW TO DELETE PSSESSIONS

When you are finished with the PSSession, use the Remove-PSSession cmdlet to delete the PSSession and to release the resources that it was using.

```
Remove-PSSession -Session $ps
```

- or -

```
Remove-PSSession -Id 1
```

To remove a PSSession from a remote computer, use the ComputerName parameter of the Remove-PSSession cmdlet.

Remove-PSSession -ComputerName Server01 -Id 1

If you do not delete the PSSession, the PSSession remains available for use until it times out.

You can also use the IdleTimeout parameter of the New-PSSessionOption cmdlet to set an expiration time for an idle PSSession. For more information, see New-PSSessionOption.

THE PSSESSION CMDLETS

Cmdlet	Description
New-PSSession	Creates a new PSSession on a local or remote computer.
Get-PSSession	Gets the PSSessions in the current session.
Remove-PSSession	Deletes the PSSessions in the current session.
Enter-PSSession	Starts an interactive session.
Exit-PSSession	Ends an interactive session.
Disconnect-PSSession	Disconnects a PSSession from the current session.
Connect-PSSession	Connects a PSSession to the current session.
Receive-PSSession	Gets the results of commands that ran in a disconnected session.

For a list of PSSession cmdlets, type:

```
get-help *-PSSession
```

FOR MORE INFORMATION

For more information about PSSessions, see about_PSSession_Details.

SEE ALSO

about_Remote
about_Remote_Disconnected_Sessions
about_Remote_Requirements
Connect-PSSession
Disconnect-PSSession
Enter-PSSession

Exit-PSSession
Get-PSSession
Invoke-Command
New-PSSession
Remove-PSSession

TOPIC

about_PSSession_Details

SHORT DESCRIPTION

Provides detailed information about Windows PowerShell sessions and the role they play in remote commands.

LONG DESCRIPTION

A session is an environment in which Windows PowerShell runs. A session is created for you whenever you start Windows PowerShell. You can create additional sessions, called "Windows PowerShell sessions" or "PSSessions" on your computer or another computer.

Unlike the sessions that Windows PowerShell creates for you, you control and manage the PSSessions that you create.

PSSessions play an important role in remote computing. When you create a PSSession that is connected to a remote computer, Windows PowerShell establishes a persistent connection to the remote computer to support the PSSession. You can use the PSSession to run a series of commands, functions, and scripts that share data.

This topic provides detailed information about sessions and PSSessions in Windows PowerShell. For basic information about the tasks that you can perform with sessions, see about_PSSessions.

ABOUT SESSIONS

Technically, a session is an execution environment in which Windows PowerShell runs. Each session includes an instance of the System.Management.Automation engine and a host program in which Windows PowerShell runs. The host can be the familiar Windows PowerShell console or another program that runs commands, such as Cmd.exe, or a program built to host Windows PowerShell, such as Windows PowerShell Integrated Scripting Environment (ISE). From a Windows perspective, a session is a Windows process on the target computer.

Each session is configured independently. It includes its own properties, its own execution policy, and its own profiles. The environment that exists when the session is created persists for its lifetime even if you change the environment on the computer. All sessions are created in a global scope, even sessions that you create in a script.

You can run only one command (or command pipeline) in a session at one time. A second command run synchronously (one at a time) waits up to four minutes for the first command to be completed. A second command run asynchronously (concurrently) fails.

ABOUT PSSESSIONS

A session is created each time that you start Windows PowerShell. And, Windows PowerShell creates temporary sessions to run individual commands. However, you can also create sessions (called "Windows PowerShell sessions" or "PSSessions") that you control and manage.

PSSessions are critical to remote commands. If you use the ComputerName parameter of the Invoke-Command or Enter-PSSession cmdlets, Windows PowerShell establishes a temporary session to run the command and then closes the session as soon as the command or the interactive session is complete.

However, if you use the New-PSSession cmdlet to create a PSSession, Windows PowerShell establishes a persistent session on the remote computer in which you can run multiple commands or interactive sessions. The PSSessions that you create remain open and available for use until you delete them or until you close the session in which they were created.

When you create a PSSession on a remote computer, the system creates a PowerShell process on the remote computer and establishes a connection from the local computer to the process on the remote computer. When you create a PSSession on the local computer, both the new process and the connections are created on the local computer.

WHEN DO I NEED A PSSESSION?

The Invoke-Command and Enter-PSSession cmdlets have both ComputerName and Session parameters. You can use either to run a remote command.

Use the ComputerName parameter to run a single command or a series of unrelated commands on one or many computers.

To run commands that share data, you need a persistent connection to the remote computer. In that case, create a PSSession, and then use the Session parameter to run commands in the PSSession.

Many other cmdlets that get data from remote computers, such as Get-Process, Get-Service, Get-EventLog, and Get-WmiObject have only a ComputerName parameter. They use technologies other than Windows PowerShell remoting to gather data remotely. These cmdlets do not have a Session parameter, but you can use the Invoke-Command cmdlet to run these commands in a PSSession.

HOW DO I CREATE A PSSESSION?

To create a PSSession, use the New-PSSession cmdlet. You can use New-PSSession to create a PSSession on a local or remote computer.

CAN I CREATE A PSSESSION ON ANY COMPUTER?

To create a PSSession that is connected to a remote computer, the computer must be configured for remoting in Windows PowerShell. The current user must be a member of the Administrators group on the remote computer, or the current user must be able to supply the credentials of a member of the Administrators group. For more information, see about_Remote_Requirements.

CAN I SEE MY PSSESSIONS IN OTHER SESSIONS?

Beginning in Windows PowerShell 3.0, the ComputerName parameter of the Get-PSSession cmdlet gets PSSessions that you created on the specified remote computers.

Active PSSessions are maintained on the remote computer (the "server-side" of a connection) and you can get them from any session on any computer.

For example, if you create a PSSession from the Server01 computer to the Server02 computer, and then switch to the Server03 computer, you can use a command like the following one to get the session.

```
Get-PSSession -ComputerName Server02
```

Even if you disconnect from the session, the session is maintained on the remote computer until you delete it or it times out.

In Windows PowerShell 2.0, you can get only the PSSessions that you have created in the current session. You cannot get PSSessions that you created in other sessions.

For more information, see Get-PSSession.

CAN I SEE THE PSSESSIONS THAT OTHERS HAVE CREATED ON MY COMPUTER?

You can get and manage only the PSSessions that others have created only if you can supply the credentials of the user who created the PSSession or the session configuration that the PSSession uses includes RunAs credentials. Otherwise, you can get, connect to, use, and manage only the PSSessions that you created.

CAN I CONNECT TO A PSSESSION FROM A DIFFERENT COMPUTER?

Beginning in Windows PowerShell 3.0, PSSessions are independent of the sessions in which they were created. Active PSSessions are maintained on the computer at the remote or "server-side" of a connection.

You can use the Disconnect-PSSession cmdlet to disconnect from a PSSession. The PSSession is disconnected from the local session, but is maintained on the remote computer. Commands continue to run in the disconnected PSSession. You can close Windows PowerShell and shut down the originating computer without interrupting the PSSession.

Then, even hours later, you can use the Get-PSSession cmdlet to get the PSSession and the Connect-PSSession cmdlet to connect to the PSSession from a new session on a different computer.

For more information, see about_Remote_Disconnected_Sessions (http://go.microsoft.com/fwlink/?LinkID=252847).

WHAT HAPPENS TO MY PSSESSION IF I MY COMPUTER STOPS?

Disconnected PSSessions are independent of the sessions in which they were created. If you disconnect a PSSession

and then close the originating computer, the PSSession is
maintained on the remote computer.

In addition, Windows PowerShell attempts to recover active
PSSessions that are disconnected unintentionally, such as
by a computer reboot, a temporary power outage or network
disruption. Windows PowerShell attempts to maintain or recover
the PSSession to an Opened state, if the originating session
is still available, or to a disconnected state if it is not.

An "active" PSSession is one that is running commands. If
a PSSession is connected (not disconnected) and commands are
running in the PSSession when the connected session closes,
Windows PowerShell attempts to maintain the PSSession on the
remote computer. However, if no commands are running in the
PSSession, Windows PowerShell closes the PSSession when the
connected session closes.

For more information, see about_Remote_Disconnected_Sessions
(http://go.microsoft.com/fwlink/?LinkID=252847).

CAN I RUN A BACKGROUND JOB IN A PSSESSION?

Yes. A background job is a command that runs asynchronously in the
background without interacting with the current session. When you submit
a command to start a job, the command returns a job object, but the job
continues to run in the background until it is complete.

To start a background job on a local computer, use the Start-Job command.
You can run the background job in a temporary connection (by using the
ComputerName parameter) or in a PSSession (by using the Session parameter).

To start a background job on a remote computer, use the Invoke-Command
cmdlet with its AsJob parameter, or use the Invoke-Command cmdlet to run a
Start-Job command on a remote computer. When using the AsJob parameter,
you can use the ComputerName or Session parameters.

When using Invoke-Command to run a Start-Job command, you must run the
command in a PSSession. If you use the ComputerName parameter, Windows
PowerShell ends the connection when the job object returns, and the job is
interrupted.

For more information, see about_Jobs.

CAN I RUN AN INTERACTIVE SESSION?

Yes. To start an interactive session with a remote computer, use the Enter-PSSession cmdlet. In an interactive session, the commands that you type run on the remote computer, just as if you typed them directly on the remote computer.

You can run an interactive session in a temporary session (by using the ComputerName parameter) or in a PSSession (by using the Session parameter). If you use a PSSession, the PSSession retains the data from previous commands, and the PSSession retains any data generated during the interactive session for use in later commands.

When you end the interactive session, the PSSession remains open and available for use.

For more information, see Enter-PSSession and Exit-PSSession.

MUST I DELETE THE PSSESSIONS?

Yes. A PSSession is a process, which is a self-contained environment that uses memory and other resources even when you are not using it. When you are finished with a PSSession, delete it. If you create multiple PSSessions, close the ones that you are not using, and maintain only the ones currently in use.

To delete PSSessions, use the Remove-PSSession cmdlet. It deletes the PSSessions and releases all of the resources that they were using.

You can also use the IdleTimeOut parameter of New-PSSessionOption to close an idle PSSession after an interval that you specify. For more information, see New-PSSessionOption.

If you save a PSSession object in a variable and then delete the PSSession or let it time out, the variable still contains the PSSession object, but the PSSession is not active and cannot be used or repaired.

ARE ALL SESSIONS AND PSSESSIONS ALIKE?

No. Developers can create custom sessions that include only selected providers and cmdlets. If a command works in one session but not in another, it might be because the session is restricted.

SEE ALSO

about_Jobs
about_PSSessions

about_Remote
about_Remote_Disconnected_Sessions
about_Remote_Requirements
Invoke-Command
New-PSSession
Get-PSSession
Remove-PSSession
Enter-PSSession
Exit-PSSession

TOPIC

about_PSSnapins

SHORT DESCRIPTION

Describes Windows PowerShell snap-ins and shows how to use and manage them.

LONG DESCRIPTION

A Windows PowerShell snap-in is a Microsoft .NET Framework assembly that contains Windows PowerShell providers and/or cmdlets. Windows PowerShell includes a set of basic snap-ins, but you can extend the power and value of Windows PowerShell by adding snap-ins that contain providers and cmdlets that you create or get from others.

When you add a snap-in, the cmdlets and providers that it contains are immediately available for use in the current session, but the change affects only the current session.

To add the snap-in to all future sessions, save it in your Windows PowerShell profile. You can also use the Export-Console cmdlet to save the snap-in names to a console file and then use it in future sessions. You can even save multiple console files, each with a different set of snap-ins.

NOTE: Windows PowerShell snap-ins (PSSnapins) are available for use in Windows PowerShell 3.0 and Windows PowerShell 2.0. They might be altered

or unavailable in subsequent versions. To package Windows PowerShell cmdlets and providers, use modules. For information about creating modules and converting snap-ins to modules, see "Writing a Windows PowerShell Module" in MSDN at http://go.microsoft.com/fwlink/?LinkID=141556.

FINDING SNAP-INS

To get a list of the Windows PowerShell snap-ins on your computer, type:

```
get-pssnapin
```

To get the snap-in for each Windows PowerShell provider, type:

```
get-psprovider | format-list name, pssnapin
```

To get a list of the cmdlets in a Windows PowerShell snap-in, type:

```
get-command -module <snap-in_name>
```

INSTALLING A SNAP-IN

The built-in snap-ins are registered in the system and added to the default session when you start Windows PowerShell. However, you have to register snap-ins that you create or obtain from others and then add the snap-ins to your session.

REGISTERING A SNAP-IN

A Windows PowerShell snap-in is a program written in a .NET Framework language that is compiled into a .dll file. To use the providers and cmdlets in a snap-in, you must first register the snap-in (add it to the registry).

Most snap-ins include an installation program (an .exe or .msi file) that registers the .dll file for you. However, if you receive a snap-in as a .dll file, you can register it on your system. For more information, see "How to Register Cmdlets, Providers, and Host Applications" in the MSDN (Microsoft Developer Network) library at http://go.microsoft.com/fwlink/?LinkID=143619.

To get all the registered snap-ins on your system or to verify that a snap-in is registered, type:

```
get-pssnapin -registered
```

ADDING THE SNAP-IN TO THE CURRENT SESSION

To add a registered snap-in to the current session, use the Add-PsSnapin cmdlet. For example, to add the Microsoft SQL Server snap-in to the

session, type:

```
add-pssnapin sql
```

After the command is completed, the providers and cmdlets in the snap-in are available in the session. However, they are available only in the current session unless you save them.

SAVING THE SNAP-INS

To use a snap-in in future Windows PowerShell sessions, add the Add-PsSnapin command to your Windows PowerShell profile. Or, export the snap-in names to a console file.

If you add the Add-PSSnapin command to your profile, it is available in all future Windows PowerShell sessions. If you export the names of the snap-ins in your session, you can use the export file only when you need the snap-ins.

To add the Add-PsSnapin command to your Windows PowerShell profile, open your profile, paste or type the command, and then save the profile. For more information, see about_Profiles.

To save the snap-ins from a session in console file (.psc1), use the Export-Console cmdlet. For example, to save the snap-ins in the current session configuration to the NewConsole.psc1 file in the current directory, type:

```
export-console NewConsole
```

For more information, see Export-Console.

OPENING WINDOWS POWERSHELL WITH A CONSOLE FILE

To use a console file that includes the snap-in, start Windows PowerShell (PowerShell.exe) from the command prompt in Cmd.exe or in another Windows PowerShell session. Use the PsConsoleFile parameter to specify the console file that includes the snap-in. For example, the following command starts Windows PowerShell with the NewConsole.psc1 console file:

```
PowerShell.exe -psconsolefile NewConsole.psc1
```

The providers and cmdlets in the snapin are now available for use in the session.

REMOVING A SNAP-IN

To remove a Windows PowerShell snap-in from the current session, use the

Remove-PsSnapin cmdlet. For example, to remove the SQL Server snap-in from the current session, type:

```
remove-pssnapin sql
```

This cmdlet removes the snap-in from the session. The snap-in is still loaded, but the providers and cmdlets that it supports are no longer available.

BUILT-IN COMMANDS

In Windows PowerShell 2.0 and in older-style host programs in Windows PowerShell 3.0 and later, the built-in commands that are installed with Windows PowerShell are packaged in snap-ins that are added automatically to every Windows PowerShell session.

Beginning in Windows PowerShell 3.0, in newer-style host programs -- those that start sessions by using the InitialSessionState.CreateDefault2 method -- the built-in commands are packaged in modules. The exception is Microsoft.PowerShell.Core, which always appears as a snap-in. The Core snap-in is included in every session by default. The built-in modules are loaded automatically on first-use.

NOTE: Remote sessions, including sessions that are started by using the New-PSSession cmdlet, are older-style sessions in which the built-in commands are packaged in snap-ins.

The following snap-ins (or modules) are installed with Windows PowerShell.

Microsoft.PowerShell.Core

Contains providers and cmdlets used to manage the basic features of Windows PowerShell. It includes the FileSystem, Registry, Alias, Environment, Function, and Variable providers and basic cmdlets like Get-Help, Get-Command, and Get-History.

Microsoft.PowerShell.Host

Contains cmdlets used by the Windows PowerShell host, such as Start-Transcript and Stop-Transcript.

Microsoft.PowerShell.Management

Contains cmdlets such as Get-Service and Get-ChildItem that are used to manage Windows-based features.

Microsoft.PowerShell.Security

Contains the Certificate provider and cmdlets used to manage Windows PowerShell security, such as Get-Acl, Get-AuthenticodeSignature, and ConvertTo-SecureString.

Microsoft.PowerShell.Utility
Contains cmdlets used to manipulate objects and data, such as Get-Member, Write-Host, and Format-List.

Microsoft.WSMan.Management
Contains the WSMan provider and cmdlets that manage the Windows Remote Management service, such as Connect-WSMan and Enable-WSManCredSSP.

LOGGING SNAP-IN EVENTS
Beginning in Windows PowerShell 3.0, you can record execution events for the cmdlets in Windows PowerShell modules and snap-ins by setting the LogPipelineExecutionDetails property of modules and snap-ins to TRUE. For more information, see about_EventLogs (http://go.microsoft.com/fwlink/?LinkID=113224).

SEE ALSO
Add-PsSnapin
Get-PsSnapin
Remove-PsSnapin
Export-Console
Get-Command
about_Profiles
about_Modules

KEYWORDS: about_Snapins, about_Snap_ins, about_Snap-ins

TOPIC

about_Quoting_Rules

SHORT DESCRIPTION
Describes rules for using single and double quotation marks in Windows PowerShell.

LONG DESCRIPTION

Quotation marks are used to specify a literal string. You can enclose a string in single quotation marks (') or double quotation marks (").

Quotation marks are also used to create a here-string. A here-string is a single-quoted or double-quoted string in which quotation marks are interpreted literally. A here-string can span multiple lines. All the lines in a here-string are interpreted as strings, even though they are not enclosed in quotation marks.

In commands to remote computers, quotation marks define the parts of the command that are run on the remote computer. In a remote session, quotation marks also determine whether the variables in a command are interpreted first on the local computer or on the remote computer.

SINGLE AND DOUBLE-QUOTED STRINGS

When you enclose a string in double quotation marks (a double-quoted string), variable names that are preceded by a dollar sign ($) are replaced with the variable's value before the string is passed to the command for processing.

For example:

```
$i = 5
"The value of $i is $i."
```

The output of this command is:

```
The value of 5 is 5.
```

Also, in a double-quoted string, expressions are evaluated, and the result is inserted in the string. For example:

```
"The value of $(2+3) is 5."
```

The output of this command is:

```
The value of 5 is 5.
```

When you enclose a string in single-quotation marks (a single-quoted string), the string is passed to the command exactly as you type it. No substitution is performed. For example:

```
$i = 5
'The value of $i is $i.'
```

The output of this command is:

```
The value $i is $i.
```

Similarly, expressions in single-quoted strings are not evaluated. They are interpreted as literals. For example:

```
'The value of $(2+3) is 5.'
```

The output of this command is:

```
The value of $(2+3) is 5.
```

To prevent the substitution of a variable value in a double-quoted string, use the backtick character (`)(ASCII 96), which is the Windows PowerShell escape character.

In the following example, the backtick character that precedes the first $i variable prevents Windows PowerShell from replacing the variable name with its value. For example:

```
$i = 5
"The value of `$i is $i."
```

The output of this command is:

```
The value $i is 5.
```

To make double-quotation marks appear in a string, enclose the entire string in single quotation marks. For example:

```
'As they say, "live and learn."'
```

The output of this command is:

```
As they say, "live and learn."
```

You can also enclose a single-quoted string in a double-quoted string. For example:

```
"As they say, 'live and learn.'"
```

The output of this command is:

```
As they say, 'live and learn.'
```

Or, double the quotation marks around a double-quoted phrase. For example:

```
"As they say, ""live and learn."""
```

The output of this command is:

```
As they say, "live and learn."
```

To include a single quotation mark in a single-quoted string, use a second consecutive single quote. For example:

```
'don''t'
```

The output of this command is:

```
don't
```

To force Windows PowerShell to interpret a double quotation mark literally, use a backtick character. This prevents Windows PowerShell from interpreting the quotation mark as a string delimiter. For example:

```
"Use a quotation mark (`") to begin a string."
```

Because the contents of single-quoted strings are interpreted literally, you cannot use the backtick character to force a literal character interpretation in a single-quoted string.

For example, the following command generates an error because Windows PowerShell does not recognize the escape character. Instead, it interprets the second quotation mark as the end of the string.

```
PS C:\> 'Use a quotation mark (`') to begin a string.'
Unexpected token ')' in expression or statement.
At line:1 char:27
+ 'Use a quotation mark (`') <<<<  to begin a string.'
```

HERE-STRINGS

The quotation rules for here-strings are slightly different.

A here-string is a single-quoted or double-quoted string in which quotation marks are interpreted literally. A here-string can span multiple lines. All the lines in a here-string are interpreted as strings even though they are not enclosed in quotation marks.

Like regular strings, variables are replaced by their values in double-quoted here-strings. In single-quoted here-strings, variables are not replaced by their values.

You can use here-strings for any text, but they are particularly useful for the following kinds of text:

-- Text that contains literal quotation marks
-- Multiple lines of text, such as the text in an HTML or XML document
-- The Help text for a script or function

A here-string can have either of the following formats, where <Enter> represents the linefeed or newline hidden character that is added when you press the ENTER key.

Double-quotes:

```
@"<Enter>
<string> [string] ...<Enter>
"@
```

Single-quotes:

```
@'<Enter>
<string> [string] ...<Enter>
'@
```

In either format, the closing quotation mark must be the first character in the line.

A here-string contains all the text between the two hidden characters. In the here-string, all quotation marks are interpreted literally. For example:

```
@"
For help, type "get-help"
"@
```

The output of this command is:

```
For help, type "get-help"
```

Using a here-string can simplify using a string in a command. For example:

```
@"
Use a quotation mark (') to begin a string.
"@
```

The output of this command is:

```
Use a quotation mark (') to begin a string.
```

In single-quoted here-strings, variables are interpreted literally and

reproduced exactly. For example:

```
@'
The $profile variable contains the path
of your Windows PowerShell profile.
'@
```

The output of this command is:

```
The $profile variable contains the path
of your Windows PowerShell profile.
```

In double-quoted here-strings, variables are replaced by their values.
For example:

```
@"
Even if you have not created a profile,
the path of the profile file is:
$profile.
"@
```

The output of this command is:

```
Even if you have not created a profile,
the path of the profile file is:
C:\Users\User01\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1.
```

Here-strings are typically used to assign multiple lines to
a variable. For example, the following here-string assigns a
page of XML to the $page variable.

```
$page = [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>
      <command:name>
           Format-Table
      </command:name>
      <maml:description>
         <maml:para>Formats the output as a table.</maml:para>
      </maml:description>
      <command:verb>format</command:verb>
      <command:noun>table</command:noun>
      <dev:version></dev:version>
</command:details>
...
```

```
    </command:command>
    "@
```

Here-strings are also a convenient format for input to the ConvertFrom-StringData cmdlet, which converts here-strings to hash tables. For more information, see ConvertFrom-StringData.

KEYWORDS

about_Here-Strings
about_Quotes
about_Quotation_Marks

SEE ALSO

about_Escape_Characters
ConvertFrom-StringData

TOPIC

about_Redirection

SHORT DESCRIPTION

Explains how to redirect output from Windows PowerShell to text files.

LONG DESCRIPTION

By default, Windows PowerShell sends its command output to the Windows PowerShell console. However, you can direct the output to a text file, and you can redirect error output to the regular output stream.

You can use the following methods to redirect output:

- Use the Out-File cmdlet, which sends command output to a text file. Typically, you use the Out-File cmdlet when you need to use its parameters, such as the Encoding, Force, Width, or NoClobber parameters.

- Use the Tee-Object cmdlet, which sends command output to a text file and then sends it to the pipeline.

- Use the Windows PowerShell redirection operators.

WINDOWS POWERSHELL REDIRECTION OPERATORS

The redirection operators enable you to send particular types of output to files and to the success output stream.

The Windows PowerShell redirection operators use the following characters to represent each output type:

*	All output
1	Success output
2	Errors
3	Warning messages
4	Verbose output
5	Debug messages
6	Informational messages

NOTE: The All (*), Warning (3), Verbose (4) and Debug (5) redirection operators were introduced in Windows PowerShell 3.0; the Information (6) redirection operator was introduced in Windows PowerShell 5.0. They do not work in earlier versions of Windows PowerShell.

The Windows PowerShell redirection operators are as follows.

Operator	Description	Example
>	Sends output to the specified file.	Get-Process > Process.txt
>>	Appends the output to the contents of the specified file.	dir *.ps1 >> Scripts.txt
2>	Sends errors to the specified file.	Get-Process none 2> Errors.txt
2>>	Appends errors to the contents of the specified file.	Get-Process none 2>> Save-Errors.txt
2>&1	Sends errors (2) and success output (1) to the success output stream.	Get-Process none, Powershell 2>&1
3>	Sends warnings to the specified file.	Write-Warning "Test!" 3> Warnings.txt

3>>	Appends warnings to the contents of the specified file.	Write-Warning "Test!" 3>> Save-Warnings.txt
3>&1	Sends warnings (3) and success output (1) to the success output stream.	function Test-Warning { Get-Process PowerShell; Write-Warning "Test!" } Test-Warning 3>&1
4>	Sends verbose output to the specified file.	Import-Module * -Verbose 4> Verbose.txt
4>>	Appends verbose output to the contents of the specified file.	Import-Module * -Verbose 4>> Save-Verbose.txt
4>&1	Sends verbose output (4) and success output (1) to the success output stream.	Import-Module * -Verbose 4>&1
5>	Sends debug messages to the specified file.	Write-Debug "Starting" 5> Debug.txt
5>>	Appends debug messages to the contents of the specified file.	Write-Debug "Saving" 5>> Save-Debug.txt
5>&1	Sends debug messages (5) and success output (1) to the success output stream.	function Test-Debug { Get-Process PowerShell Write-Debug "PS" } Test-Debug 5>&1
6>	Sends informational messages to a specified file	Write-Information "Here they are" 6> Info.txt
6>>	Appends informational messages to the contents of a specified file.	Write-Information "Nothing found" 6>> Info.txt
6>&1	Sends informational messages and success output to the success output stream.	function Test-Info { Get-Process P* Write-Information "Here you go" } Test-Info 6>&1
*>	Sends all output types	function Test-Output

to the specified file. { Get-Process PowerShell, none
                        Write-Warning "Test!"
*>> Appends all output types Write-Verbose "Test Verbose"
to the contents of the Write-Debug "Test Debug" }
specified file.
                        Test-Output *> Test-Output.txt
*>&1 Sends all output types Test-Output *>> Test-Output.txt
(*) to the success output Test-Output *>&1
stream.

The syntax of the redirection operators is as follows:

<input> <operator> [<path>\]<file>

If the specified file already exists, the redirection operators that do not append data (> and n>) overwrite the current contents of the file without warning. However, if the file is a read-only, hidden, or system file, the redirection fails. The append redirection operators (>> and n>>) do not write to a read-only file, but they append content to a system or hidden file.

To force the redirection of content to a read-only, hidden, or system file, use the Out-File cmdlet with its Force parameter. When you are writing to files, the redirection operators use Unicode encoding. If the file has a different encoding, the output might not be formatted correctly. To redirect content to non-Unicode files, use the Out-File cmdlet with its Encoding parameter.

SEE ALSO

Out-File
Tee-Object
about_Operators
about_Command_Syntax
about_Path_Syntax

TOPIC

about_Ref

SHORT DESCRIPTION

Describes how to create and use a reference variable type.

LONG DESCRIPTION

You can use the reference variable type to permit a method to change the value of a variable that is passed to it.

When the [ref] type is associated with an object, it returns a reference to that object. If the reference is used with a method, the method can refer to the object that was passed to it. If the object is changed within the method, the change appears as a change in the value of the variable when control returns to the calling method.

To use referencing, the parameter must be a reference variable. If it is not, an InvalidArgument exception is thrown.

The parameters used in method invocations must match the type required by the methods.

Examples:

```
PS> function swap([ref]$a,[ref]$b)
>> {
>>    $a.value,$b.value = $b.value,$a.value
>> }

PS> $a = 1
PS> $b = 10
PS> $a,$b
1
10
PS> swap ([ref]$a) ([ref]$b)
PS> $a,$b
10
1

PS C:\ps-test> function double
>> {
>>    param ([ref]$x) $x.value = $x.value * 2
>> }

PS C:> $number = 8
PS C:> $number
```

8
PS C> double ([ref]$number)
PS C> $number
16

The variable must be a reference variable.

PS C:\ps-test> double $number
double : Reference type is expected in argument.
At line:1 char:7
+ double <<<< $number

SEE ALSO
about_Variables
about_Environment_Variables
about_Functions
about_Script_Blocks

TOPIC

about_Regular_Expressions

SHORT DESCRIPTION
Describes regular expressions in Windows PowerShell.

LONG DESCRIPTION
Windows PowerShell supports the following regular expression characters.

Format	Logic	Example
value	Matches exact characters anywhere in the original value.	"book" -match "oo"

.	Matches any single character.	"copy" -match "c..y"
[value]	Matches at least one of the characters in the brackets.	"big" -match "b[iou]g"
[range]	Matches at least one of the characters within the range. The use of a hyphen (–) allows you to specify an adjacent character.	"and" -match "[a-e]nd"
[^]	Matches any characters except those in brackets.	"and" -match "[^brt]nd"
^	Matches the beginning characters.	"book" -match "^bo"
$	Matches the end characters.	"book" -match "ok$"
*	Matches any instances of the preceding character.	"baggy" -match "g*"
?	Matches zero or one instance of the preceding character.	"baggy" -match "g?"
\	Matches the character that follows as an escaped character.	"Try$" -match "Try\$"

Windows PowerShell supports the character classes available in Microsoft .NET Framework regular expressions.

Format	Logic	Example
\p{name}	Matches any character in the named character class specified by {name}. Supported names are Unicode groups and block ranges such as Ll, Nd, Z, IsGreek, and IsBoxDrawing.	"abcd defg" -match "\p{Ll}+"
\P{name}	Matches text not included in the groups and block ranges specified in {name}.	1234 -match "\P{Ll}+"
\w	Matches any word character.	"abcd defg" -match "\w+"

	Equivalent to the Unicode character categories [\p{Ll} \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].	(this matches abcd)
\W	Matches any nonword character. Equivalent to the Unicode categories [^\p{Ll}\p{Lu}\p{Lt} \p{Lo}\p{Nd}\p{Pc}].	"abcd defg" -match "\W+" (This matches the space)
\s	Matches any white-space character. Equivalent to the Unicode character categories [\f\n\r\t\v\x85\p{Z}].	"abcd defg" -match "\s+"
\S	Matches any non-white-space character. Equivalent to the Unicode character categories [^\f\n\r\t\v\x85\p{Z}].	"abcd defg" -match "\S+"
\d	Matches any decimal digit. Equivalent to \p{Nd} for Unicode and [0-9] for non-Unicode behavior.	12345 -match "\d+"
\D	Matches any nondigit. Equivalent to \P{Nd} for Unicode and [^0-9] for non-Unicode behavior.	"abcd" -match "\D+"

Windows PowerShell supports the quantifiers available in .NET Framework regular expressions. The following are some examples of quantifiers.

Format	Logic	Example
*	Specifies zero or more matches; for example, \w* or (abc)*. Equivalent to {0,}.	"abc" -match "\w*"
+	Matches repeating instances of the preceding characters.	"xyxyxy" -match "xy+"
?	Specifies zero or one matches;	"abc" -match "\w?"

for example, \w? or (abc)?.
Equivalent to {0,1}.

{n} Specifies exactly n matches; "abc" -match "\w{2}"
for example, (pizza){2}.

{n,} Specifies at least n matches; "abc" -match "\w{2,}"
for example, (abc){2,}.

{n,m} Specifies at least n, but no "abc" -match "\w{2,3}"
more than m, matches.

All the comparisons shown in the preceding table evaluate to true.

Notice that the escape character for regular expressions, a backslash (\), is different from the escape character for Windows PowerShell. The escape character for Windows PowerShell is the backtick character (`) (ASCII 96).

For more information, see the "Regular Expression Language Elements" topic in the Microsoft Developer Network (MSDN) library at http://go.microsoft.com/fwlink/?LinkId=133231.

SEE ALSO

about_Comparison_Operators
about_Operators

TOPIC

about_Remote

SHORT DESCRIPTION

Describes how to run remote commands in Windows PowerShell.

LONG DESCRIPTION

You can run remote commands on a single computer or on multiple computers by using a temporary or persistent connection. You can also start an interactive session with a single remote computer.

This topic provides a series of examples to show you how to run different types of remote command. After you try these basic commands, read the Help topics that describe each cmdlet that is used in these commands. The topics provide the details and explain how you can modify the commands to meet your needs.

Note: To use Windows PowerShell remoting, the local and remote computers must be configured for remoting. For more information, see about_Remote_Requirements.

HOW TO START AN INTERACTIVE SESSION (ENTER-PSSESSION)

The easiest way to run remote commands is to start an interactive session with a remote computer.

When the session starts, the commands that you type run on the remote computer, just as though you typed them directly on the remote computer. You can connect to only one computer in each interactive session.

To start an interactive session, use the Enter-PSSession cmdlet. The following command starts an interactive session with the Server01 computer:

Enter-PSSession Server01

The command prompt changes to indicate that you are connected to the Server01 computer.

```
Server01\PS>
```

Now, you can type commands on the Server01 computer.

To end the interactive session, type:

Exit-PSSession

For more information, see Enter-PSSession.

HOW TO USE CMDLETS THAT HAVE A COMPUTERNAME PARAMETER TO GET REMOTE DATA

Several cmdlets have a ComputerName parameter that lets you get objects from remote computers.

Because these cmdlets do not use WS-Management-based Windows PowerShell remoting, you can use the ComputerName parameter of these cmdlets on any computer that is running Windows PowerShell. The computers do not have to be configured for Windows PowerShell remoting, and the computers do not have to meet the system requirements for remoting.

The following cmdlets have a ComputerName parameter:

Clear-EventLog Limit-EventLog
Get-Counter New-EventLog
Get-EventLog Remove-EventLog
Get-HotFix Restart-Computer
Get-Process Show-EventLog
Get-Service Stop-Computer
Get-WinEvent Test-Connection
Get-WmiObject Write-EventLog

For example, the following command gets the services on the Server01 remote computer:

```
Get-Service -ComputerName Server01
```

Typically, cmdlets that support remoting without special configuration have a ComputerName parameter and do not have a Session parameter. To find these cmdlets in your session, type:

```
Get-Command | where { $_.Parameters.Keys -contains "ComputerName" -and $_.Parameters.Keys -NotContains "Session"}
```

HOW TO RUN A REMOTE COMMAND

To run other commands on remote computers, use the Invoke-Command cmdlet.

To run a single command or a few unrelated commands, use the ComputerName parameter of Invoke-Command to specify the remote computers. Use the ScriptBlock parameter to specify the command.

For example, the following command runs a Get-Culture command on the Server01 computer.

```
Invoke-Command -ComputerName Server01 -ScriptBlock {Get-Culture}
```

The ComputerName parameter is designed for situation in which you run a single command or several unrelated commands on one or many computers. To establish a persistent connection to a remote computer, use the Session parameter.

HOW TO CREATE A PERSISTENT CONNECTION (PSSESSION)

When you use the ComputerName parameter of the Invoke-Command cmdlet, Windows PowerShell establishes a connection just for the command. Then, it closes the connection when the command is complete. Any variables or functions that are defined in the command are lost.

To create a persistent connection to a remote computer, use the New-PSSession cmdlet. For example, the following command creates PSSessions on the Server01 and Server02 computers and then saves the PSSessions in the $s variable.

```
$s = New-PSSession -ComputerName Server01, Server02
```

HOW TO RUN COMMANDS IN A PSSESSION

With a PSSession, you can run a series of remote commands that share data, like functions, aliases, and the values of variables. To run commands in a PSSession, use the Session parameter of the Invoke-Command cmdlet.

For example, the following command uses the Invoke-Command cmdlet to run a Get-Process command in the PSSessions on the Server01 and Server02 computers. The command saves the processes in a $p variable in each PSSession.

```
Invoke-Command -Session $s -ScriptBlock {$p = Get-Process}
```

Because the PSSession uses a persistent connection, you can run another command in the same PSSession that uses the $p variable. The following command counts the number of processes saved in $p.

```
Invoke-Command -Session $s -ScriptBlock {$p.count}
```

HOW TO RUN A REMOTE COMMAND ON MULTIPLE COMPUTERS

To run a remote command on multiple computers, type all of the computer names in the value of the ComputerName parameter of Invoke-Command. Separate the names with commas.

For example, the following command runs a Get-Culture command on three computers:

```
Invoke-Command -ComputerName S1, S2, S3 -ScriptBlock {Get-Culture}
```

You can also run a command in multiple PSSessions. The following commands create PSSessions on the Server01, Server02, and Server03 computers and then run a Get-Culture command in each of the PSSessions.

```
    $s = New-PSSession -ComputerName S1, S2, S3
Invoke-Command -Session $s -ScriptBlock {Get-Culture}
```

To include the local computer list of computers, type the name of the local computer, type a dot (.), or type "localhost".

```
Invoke-Command -ComputerName S1, S2, S3, localhost -ScriptBlock {Get-Culture}
```

HOW TO RUN A SCRIPT ON REMOTE COMPUTERS

To run a local script on remote computers, use the FilePath parameter of Invoke-Command.

For example, the following command runs the Sample.ps1 script on the S1 and S2 computers:

```
    Invoke-Command -ComputerName S1, S2 -FilePath C:\Test\Sample.ps1
```

The results of the script are returned to the local computer. You do not need to copy any files.

HOW TO STOP A REMOTE COMMAND

To interrupt a command, press CTRL+C. The interrupt request is passed to the remote computer where it terminates the remote command.

FOR MORE INFORMATION

-- For information about the system requirements for remoting,
see about_Remote_Requirements.

-- For help in formatting remote output, see about_Remote_Output.

-- For information about how remoting works, how to manage remote
data, special configurations, security issues, and other frequently
asked questions, see about_Remote_FAQ.

-- For help in resolving remoting errors, see about_Remote_Troubleshooting.

-- For information about PSSessions and persistent connections, see
about_PSSessions.

-- For information about Windows PowerShell background jobs, see
about_Jobs.

KEYWORDS
about_Remoting

SEE ALSO
about_PSSessions
about_Remote_Disconnected_Sessions
about_Remote_Requirements
about_Remote_FAQ
about_Remote_TroubleShooting
about_Remote_Variables
Enter-PSSession
Invoke-Command
New-PSSession

TOPIC

about_Remote_Disconnected_Session

SHORT DESCRIPTION

Explains how to disconnect from and reconnect to a PSSession

LONG DESCRIPTION

Beginning in Windows PowerShell 3.0, you can disconnect from a PSSession and reconnect to the PSSession at a later time on the same computer or a different computer. The session state is maintained and commands in the PSSession continue to run while the session is disconnected.

The Disconnected Sessions feature is available only when the computer at the remote end of a connection is running Windows PowerShell 3.0 or a later version of Windows PowerShell.

The Disconnected Sessions feature allows you to close the session in which a PSSession was created, and even close Windows PowerShell, and shut down the computer, without disrupting commands running in the PSSession. It is especially useful for running commands that take an extended time to complete, and it provides the time and device flexibility that IT professionals require.

NOTE: You cannot disconnect from an interactive session that is started by using the Enter-PSSession cmdlet.

You can use Disconnected Sessions to manage PSSessions that were disconnected unintentionally as the result of a computer or network outage.

In real-world use, the Disconnected Sessions feature allows you to begin solving a problem, turn your attention to a higher priority issue, and then resume work on the solution, even on a different computer in a different location.

DISCONNECTED SESSION CMDLETS

The following cmdlets support the Disconnected Sessions feature:

Disconnect-PSSession: Disconnects a PSSession.

Connect-PSSession: Connects to a disconnected PSSession.

Receive-PSSession: Gets the results of commands that ran in disconnected sessions.

Get-PSSession: Gets PSSessions on the local computer or on remote computers.

Invoke-Command: InDisconnectedSession parameter creates

a PSSession and disconnects immediately.

HOW THE DISCONNECTED SESSIONS FEATURE WORKS

Beginning in Windows PowerShell 3.0, PSSessions are independent of the sessions in which they are created. Active PSSession are maintained on the remote computer or "server side" of the connection, even if the session in which PSSession was created is closed and the originating computer is shut down or disconnected from the network.

In Windows PowerShell 2.0, the PSSession is deleted from the remote computer when it is disconnected from the originating session or the session in which it was created ends.

When you disconnect a PSSession, the PSSession remains active and is maintained on the remote computer. The session state changes from Running to Disconnected. You can reconnect to a disconnected PSSession from the current session or from a different session on the same computer, or from a different computer. The remote computer that maintains the session must be running and be connected to the network.

Commands in a disconnected PSSession continue to run uninterrupted on the remote computer until the command completes or the output buffer fills. To prevent a full output buffer from suspending a command, use the OutputBufferingMode parameter of the Disconnect-PSSession, New-PSSessionOption, or New-PSTransportOption cmdlets.

Disconnected sessions are maintained in the disconnected state on the remote computer. They are available for you to reconnect, until you delete the PSSession, such as by using the Remove-PSSession cmdlet, or until the idle timeout of the PSSession expires. You can adjust the idle timeout of a PSSession by using the IdleTimeoutSec or IdleTimeout parameters of the Disconnect-PSSession, New-PSSessionOption, or New-PSTransportOption cmdlets.

Another user can connect to PSSessions that you created, but only if they can supply the credentials that were used to create the session, or use the RunAs credentials of the session configuration.

HOW TO GET PSSESSIONS

Beginning in Windows PowerShell 3.0, the Get-PSSession cmdlet gets

PSSessions on the local computer and remote computers. It can also get PSSessions that were created in the current session.

To get PSsessions on the local computer or remote computers, use the ComputerName or ConnectionUri parameters. Without parameters, Get-PSSession gets PSSession that were created in the local session, regardless of where they terminate.

When getting PSSessions, remember to look for them on the computer on which they are maintained, that is, the remote or "server-side" computer.

For example, if you create a PSSession to the Server01 computer, get the session from the Server01 computer. If you create a PSSession from another computer to the local computer, get the session from the local computer.

The following command sequence shows how Get-PSSession works.

The first command creates a session to the Server01 computer. The session resides on the Server01 computer.

```
PS C:\ps-test> New-PSSession -ComputerName Server01

 Id Name          ComputerName    State         ConfigurationName     Availability
 -- ----          ------------    -----         -----------------     ------------
  2 Session2       Server01        Opened        Microsoft.PowerShell     Available
```

To get the session, use the ComputerName parameter of Get-PSSession with a value of Server01.

```
PS C:\ps-test> Get-PSSession -ComputerName Server01

 Id Name          ComputerName    State         ConfigurationName     Availability
 -- ----          ------------    -----         -----------------     ------------
  2 Session2       Server01        Opened        Microsoft.PowerShell     Available
```

If the value of the ComputerName parameter of Get-PSSession is localhost, Get-PSSession gets PSSessions that terminate at and are maintained on the local computer. It does not get PSSessions on the Server01 computer, even if they were started on the local computer.

```
PS C:\ps-test> Get-PSSession -ComputerName localhost
PS C:\ps-test>
```

To get sessions that were created in the current session, use the Get-PSSession cmdlet without parameters. This command gets the PSSession that was created in the current session and connects to the Server01 computer.

```
PS C:\ps-test> Get-PSSession

 Id Name          ComputerName    State         ConfigurationName     Availability
 -- ----          ------------    -----         -----------------     ------------
  2 Session2       Server01        Opened        Microsoft.PowerShell     Available
```

HOW TO DISCONNECT SESSIONS

To disconnect a PSSession use the Disconnect-PSSession cmdlet. To identify the PSSession, use the Session parameter, or pipe a PSSession from the New-PSSession or Get-PSSession cmdlets to Disconnect-PSSession.

The following command disconnects the PSSession to the Server01 computer. Notice that the value of the State property is Disconnected and the Availability is None.

```
PS C:\> Get-PSSession -ComputerName Server01 | Disconnect-PSSession

 Id Name          ComputerName    State         ConfigurationName     Availability
 -- ----          ------------    -----         -----------------     ------------
  2 Session2       Server01        Disconnected  Microsoft.PowerShell        None
```

To create a disconnected session, use the InDisconnectedSession parameter of the Invoke-Command cmdlet. It creates a session, starts the command, and disconnects immediately, before the command can return any output.

The following command runs a Get-WinEvent command in a disconnected session on the Server02 remote computer.

```
PS C:\> Invoke-Command -ComputerName Server02 -InDisconnectedSession `
      -ScriptBlock {Get-WinEvent -LogName "Windows PowerShell"}

 Id Name          ComputerName    State         ConfigurationName     Availability
 -- ----          ------------    -----         -----------------     ------------
  4 Session3       Server02        Disconnected  Microsoft.PowerShell        None
```

HOW TO CONNECT TO DISCONNECTED SESSIONS

You can connect to any available disconnected PSSession from the session in which you created the PSSession or from other sessions on the local computer or other computers.

You can create a PSSession, run commands in the PSSession, disconnect from the PSSession, close Windows PowerShell, and shut down the computer. Hours later, you can open a different computer, get the PSSession, connect to it, and get the results of commands that ran in the PSSession while it was disconnected. Then you can run more commands in the session.

To connect a disconnected PSSession, use the Connect-PSSession cmdlet. Use the ComputerName or ConnectionUri parameters to identify the PSSession, or pipe a PSSession from Get-PSSession to Connect-PSSession.

The following command gets the sessions on the Server02 computer. The output includes two disconnected sessions, both of which are available.

```
PS C:\> Get-PSSession -ComputerName Server02

 Id Name          ComputerName    State         ConfigurationName     Availability
 -- ----          ------------    -----         -----------------     ------------
  2 Session2       juneb-srv8320   Disconnected  Microsoft.PowerShell        None
  4 Session3       juneb-srv8320   Disconnected  Microsoft.PowerShell        None
```

The following command connects to Session2. The PSSession is now open and available.

```
PS C:> Connect-PSSession -ComputerName Server02 -Name Session2

 Id Name          ComputerName    State         ConfigurationName     Availability
 -- ----          ------------    -----         -----------------     ------------
  2 Session2       juneb-srv8320   Opened        Microsoft.PowerShell     Available
```

HOW TO GET THE RESULTS

To get the results of commands that ran in a disconnected PSSession, use the Receive-PSSession cmdlet.

You can use Receive-PSSession in addition to, or instead of, using the Connect-PSSession cmdlet. If the session is already reconnected, Receive-PSSession gets the results of commands that ran when the session was disconnected. If the PSSession is still disconnected, Receive-PSSession

connects to it and then gets the results of commands that ran while it was disconnected.

Receive-PSSession can return the results in a job (asynchronously) or to the host program (synchronously). Use the OutTarget parameter to select Job or Host. Host is the default value. However, if the command that is being received was started in the current session as a job, it is returned as a job by default.

The following command uses the Receive-PSSession cmdlet to connect to the PSSession on the Server02 computer and get the results of the Get-WinEvent command that ran in the Session3 session. The command uses the OutTarget parameter to get the results in a job.

```
PS C:\> Receive-PSSession  -ComputerName Server02 -Name Session3 -OutTarget Job

Id    Name          PSJobTypeName   State       HasMoreData     Location
--    ----          -------------   -----       -----------     --------
3     Job3          RemoteJob       Running     True            Server02
```

To get the results of the job, use the Receive-Job cmdlet.

```
PS C:\ps-test> Get-Job | Receive-Job -Keep


   ProviderName: PowerShell

TimeCreated           Id LevelDisplayName Message     PSComputerName
-----------           -- ---------------- -------     --------------
5/14/2012 7:26:04 PM   400 Information     Engine stat Server02
5/14/2012 7:26:03 PM   600 Information     Provider "W Server02
5/14/2012 7:26:03 PM   600 Information     Provider "C Server02
5/14/2012 7:26:03 PM   600 Information     Provider "V Server02
```

STATE AND AVAILABILITY

The State and Availability properties of a disconnected PSSession tell you whether the session is available for you to reconnect to it.

When a PSSession is connected to the current session, its state is Opened and its availability is Available. When you disconnect from the PSSession, the PSSession state is Disconnected and its Availability is none.

However, the value of the State property is relative to the current session. Therefore, a value of Disconnected means that the PSSession is not connected to the current session. However, it does not mean that the PSSession is disconnected from all sessions. It might be connected to a different session.

To determine whether you can connect or reconnect to the PSSession, use the Availability property. An Availability value of None indicates that you can connect to the session. A value of Busy indicates that you cannot connect to the PSSession because it is connected to another session.

The following example is run in two sessions (Windows PowerShell console windows) on the same computer. Note the changing values of the State and Availability properties in each session as the PSSession is disconnected and reconnected.

```
#Session 1:
PS C:\> New-PSSession -ComputerName Server30 -Name Test

Id Name      ComputerName    State       ConfigurationName     Availability
-- ----      ------------    -----       -----------------     ------------
1  Test      Server30        Opened      Microsoft.PowerShell     Available

#Session 2:
PS C:\> Get-PSSession -ComputerName Server30 -Name Test

Id Name      ComputerName    State       ConfigurationName     Availability
-- ----      ------------    -----       -----------------     ------------
1 Test       Server30        Disconnected  Microsoft.PowerShell        Busy


#Session 1
PS C:\> Get-PSSession -ComputerName Server30 -Name Test | Disconnect-PSSession

Id Name      ComputerName    State       ConfigurationName     Availability
-- ----      ------------    -----       -----------------     ------------
 1 Test      Server30        Disconnected  Microsoft.PowerShell        None

#Session 2
PS C:\> Get-PSSession -ComputerName Server30

Id Name      ComputerName    State       ConfigurationName     Availability
-- ----      ------------    -----       -----------------     ------------
 1 Test      Server30        Disconnected  Microsoft.PowerShell        None
```

```
#Session 2
PS C:\> Connect-PSSession -ComputerName Server01 -Name Test

Id Name      ComputerName    State        ConfigurationName     Availability
-- ----      ------------    -----        -----------------     ------------
3 Test       Server30        Opened       Microsoft.PowerShell     Available

#Session 1
PS C:\> Get-PSSession -ComputerName Server30

Id Name      ComputerName    State        ConfigurationName     Availability
-- ----      ------------    -----        -----------------     ------------
 1 Test      Server30        Disconnected  Microsoft.PowerShell        Busy
```

IDLE TIMEOUT

Disconnected sessions are maintained on the remote computer until you delete them, such as by using the Remove-PSSession cmdlet, or they time out. The IdleTimeout property of a PSSession determines how long a disconnected session is maintained before it is deleted.

PSSessions are idle when the "heartbeat thread" receives no response. Disconnecting a session makes it idle and starts the Idle Timeout clock, even if commands are still running in the disconnected session. Windows PowerShell considers disconnected sessions to be active, but idle.

When creating and disconnecting sessions, verify that the idle timeout in the PSSession is long enough to maintain the session for your needs, but not so long that it consumes unnecessary resources on the remote computer.

The IdleTimeoutMs property of the session configuration determines the default idle timeout of sessions that use the session configuration. You can override the default value, but the value that you use cannot exceed the MaxIdleTimeoutMs property of the session configuration.

To find the value of the IdleTimeoutMs and MaxIdleTimeoutMs values of a session configuration, use the following command format.

```
Get-PSSessionConfiguration | Format-Table Name, IdleTimeoutMs, MaxIdleTimeoutMs
```

You can override the default value in the session configuration and set the idle timeout of a PSSession when you create a PSSession and when you disconnect.

If you are a member of the Administrators group on the remote computer, you can also create and change the IdleTimeoutMs and MaxIdleTimeoutMs properties of session configurations.

NOTES:

The idle timeout value of session configurations and session options is in milliseconds. The idle timeout value of sessions and session configuration options is in seconds.

You can set the idle timeout of a PSSession when you create the PSSession (New-PSSession, Invoke-Command) and when you disconnect from it (Disconnect-PSSession). However, you cannot change the IdleTimeout value when you connect to the PSSession (Connect-PSSession) or get results (Receive-PSSession).

The Connect-PSSession and Receive-PSSession cmdlets have a SessionOption parameter that takes a SessionOption object, such as one returned by the New-PSSessionOption cmdlet. However, the IdleTimeout value in SessionOption object and the IdleTimeout value in the $PSSessionOption preference variable do not change the value of the IdleTimeout of the PSSession in a Connect-PSSession or Receive-PSSession command.

-- To create a PSSession with a particular idle timeout value, create a $PSSessionOption preference variable. Set the value of the IdleTimeout property to the desired value (in milliseconds).

When you create PSSessions, the values in $PSSessionOption variable take precedence over the values in the session configuration.

For example, this command sets an idle timeout of 48 hours.

```
PS C:\> $PSSessionOption = New-PSSessionOption -IdleTimeoutMSec 172800000
```

-- To create a PSSession with a particular idle timeout value, use the IdleTimeoutMSec parameter of the New-PSSessionOption cmdlet. Then, use the session option in the value of the SessionOption parameter of the New-PSSession or Invoke-Command cmdlets.

The values set when creating the session take precedence over the

values set in the $PSSessionOption preference variable and the
session configuration.

For example:

```
PS C:\> $o = New-PSSessionOption -IdleTimeoutMSec 172800000
PS C:\> New-PSSession -SessionOption $o
```

-- To change a the idle timeout of a PSSession when disconnecting,
use the IdleTimeoutSec parameter of the Disconnect-PSSession
cmdlet.

For example:

```
PS C:\> Disconnect-PSSession -IdleTimeoutSec 172800
```

-- To create a session configuration with a particular idle timeout
and maximum idle timeout, use the IdleTimeoutSec and MaxIdleTimeoutSec
parameters of the New-PSTransportOption cmdlet. Then, use the
transport option in the value of the TransportOption parameter
of Register-PSSessionConfiguration.

For example:

```
PS C:\> $o = New-PSTransportOption -IdleTimeoutSec 172800 -MaxIdleTimeoutSec 259200
PS C:\> Register-PSSessionConfiguration -Name Test -TransportOption $o
```

-- To change the default idle timeout and maximum idle timeout of
a session configuration, use the IdleTimeoutSec and MaxIdleTimeoutSec
parameters of the New-PSTransportOption cmdlet. Then, use the
transport option in the value of the TransportOption parameter
of Set-PSSessionConfiguration.
For example:

```
PS C:\> $o = New-PSTransportOption -IdleTimeoutSec 172800 -MaxIdleTimeoutSec 259200
PS C:\> Set-PSSessionConfiguration -Name Test -TransportOption $o
```

OUTPUT BUFFERING MODE

The output buffering mode of a PSSession determines how command
output is managed when the output buffer of the PSSession is full.

In a disconnected session, the output buffering mode effectively determines whether the command continues to run while the session is disconnected.

Valid values:

-- Block: When the output buffer is full, execution is suspended until the buffer is clear.

-- Drop: When the output buffer is full, execution continues. As new output is generated, the oldest output is discarded.

Block, the default value, preserves data, but might interrupt the command

A value of Drop allows the command to complete, although data might be lost. When using the Drop value, redirect the command output to a file on disk. This value is recommended for disconnected sessions.

The OutputBufferingMode property of the session configuration determines the default output buffering mode of sessions that use the session configuration.

To find the value of the OutputBufferingMode of a session configuration, use the following command formats.

```
(Get-PSSessionConfiguration <ConfigurationName>).OutputBufferingMode
```

-or-

```
Get-PSSessionConfiguration | Format-Table Name, OutputBufferingMode
```

You can override the default value in the session configuration and set the output buffering mode of a PSSession when you create a PSSession, when you disconnect, and when you reconnect.

If you are a member of the Administrators group on the remote computer, you can also create and change the output buffering mode of session configurations.

-- To create a PSSession with an output buffering mode of Drop, create a $PSSessionOption preference variable in which the value of the OutputBufferingMode property is Drop.

When you create PSSessions, the values in $PSSessionOption variable take precedence over the values in the session configuration.

For example:

```
PS C:\> $PSSessionOption = New-PSSessionOption -OutputBufferingMode Drop
```

-- To create a PSSession with an output buffering mode of Drop, use the OutputBufferingMode parameter of the New-PSSessionOption cmdlet to create a session option with a value of Drop. Then, use the session option in the value of the SessionOption parameter of the New-PSSession or Invoke-Command cmdlets.

The values set when creating the session take precedence over the values set in the $PSSessionOption preference variable and the session configuration.

For example:

```
PS C:\> $o = New-PSSessionOption -OutputBufferingMode Drop
PS C:\> New-PSSession -SessionOption $o
```

-- To change a the output buffering mode of a PSSession when disconnecting, use the OutputBufferingMode parameter of the Disconnect-PSSession cmdlet.

For example:

```
PS C:\> Disconnect-PSSession -OutputBufferingMode Drop
```

-- To change a the output buffering mode of a PSSession when reconnecting, use the OutputBufferingMode parameter of the New-PSSessionOption cmdlet to create a session option with a value of Drop. Then, use the session option in the value of the SessionOption parameter of Connect-PSSession or Receive-PSSession.

For example:

```
PS C:\> $o = New-PSSessionOption -OutputBufferingMode Drop
```

PS C:\> Connect-PSSession -Cn Server01 -Name Test -SessionOption $o

-- To create a session configuration with a default output buffering mode of Drop, use the OutputBufferingMode parameter of the New-PSTransportOption cmdlet to create a transport option object with a value of Drop. Then, use the transport option in the value of the TransportOption parameter of Register-PSSessionConfiguration.

For example:

```
PS C:\> $o = New-PSTransportOption -OutputBufferingMode Drop
PS C:\> Register-PSSessionConfiguration -Name Test -TransportOption $o
```

-- To change the default output buffering mode of a session configuration, use the OutputBufferingMode parameter of the New-PSTransportOption cmdlet to create a transport option with a value of Drop. Then, use the Transport option in the value of the SessionOption parameter of Set-PSSessionConfiguration.

For example:

```
PS C:\> $o = New-PSTransportOption -OutputBufferingMode Drop
PS C:\> Set-PSSessionConfiguration -Name Test -TransportOption $o
```

DISCONNECTING LOOPBACK SESSIONS

"Loopback sessions" or "local sessions" are PSSessions that originate and terminate on the same computer. Like other PSSessions, active loopback sessions are maintained on the computer on the remote end of the connection (the local computer), so you can disconnect from and reconnect to loopback sessions.

By default, loopback sessions are created with a network security token that does not permit commands run in the session to access other computers. You can reconnect to loopback sessions that have a network security token from any session on the local computer or a remote computer.

However, if you use the EnableNetworkAccess parameter of the New-PSSession, Enter-PSSession, or Invoke-Command cmdlet, the loopback session is created with an interactive security token. The interactive token enables commands that run in the loopback session to get data from other computers.

You can disconnect loopback sessions with interactive tokens and then reconnect to them from the same session or a different session on the same computer. However, to prevent malicious access, you can reconnect to loopback sessions with interactive tokens only from the computer on which they were created.

WAITING FOR JOBS IN DISCONNECTED SESSIONS

The Wait-Job cmdlet waits until a job completes and then returns to the command prompt or the next command. By default, Wait-Job returns if the session in which a job is running is disconnected. To direct the Wait-Job cmdlet to wait until the session is reconnected (in the Opened state), use the Force parameter. For more information, see Wait-Job.

ROBUST SESSIONS AND UNINTENTIONAL DISCONNECTION

Occasionally, a PSSession might be disconnected unintentionally due to a computer failure or network outage. Windows PowerShell attempts to recover the PSSession, but its success depends upon the severity and duration of the cause.

The state of an unintentionally disconnected PSSession might be Broken or Closed, but it might also be Disconnected. If the value of State is Disconnected, you can use the same techniques to manage the PSSession as you would if the session were disconnected intentionally. For example, you can use the Connect-PSSession cmdlet to reconnect to the session and the Receive-PSSession cmdlet to get results of commands that ran while the session was disconnected.

If you close (exit) the session in which a PSSession was created while commands are running in the PSSession, Windows PowerShell maintains the PSSession in the Disconnected state on the remote computer. If you close (exit) the session in which a PSSession was created, but no commands are running in the PSSession, Windows PowerShell does not attempt to maintain the PSSession.

KEYWORDS

about_Disconnected_Sessions

SEE ALSO

about_Jobs
about_Remote
about_Remote_Variables

about_PSSessions
about_Session_Configurations
Disconnect-PSSession
Connect-PSSession
Get-PSSession
Receive-PSSession
Invoke-Command

TOPIC

about_Remote_FAQ

SHORT DESCRIPTION

Contains questions and answers about running remote commands in Windows PowerShell.

LONG DESCRIPTION

When you work remotely, you type commands in Windows 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 Windows PowerShell remoting, the remote computer must be configured for remoting. For more information, see about_Remote_Requirements.

MUST BOTH COMPUTERS HAVE WINDOWS POWERSHELL INSTALLED?

Yes. To work remotely, the local and remote computers must have Windows 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 Windows 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 Windows PowerShell engine on the remote computer, and it runs in the Windows PowerShell client on the remote computer. The command results are sent back to the local computer and appear in the Windows PowerShell session on the local computer.

To transmit the commands and receive the output, Windows PowerShell uses the WS-Management protocol. For information about the WS-Management protocol, see "WS-Management Protocol" in the MSDN (Microsoft Developer Network) library at http://go.microsoft.com/fwlink/?LinkId=144634.

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 WINDOWS POWERSHELL REMOTING SECURE?

When you connect to a remote computer, the system uses the user name 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 parameters 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 WINDOWS POWERSHELL REMOTING?

No. Several cmdlets have a ComputerName parameter that lets you get objects from the remote computer.

These cmdlets do not use Windows PowerShell remoting. So, you can use them on any computer that is running Windows PowerShell, even if the computer is not configured for Windows PowerShell remoting or if the computer does not meet the requirements for Windows PowerShell remoting.

These cmdlets include the following cmdlets:

Get-Process
Get-Service
Get-WinEvent
Get-EventLog
Get-WmiObject
Test-Connection

To find all the cmdlets with a ComputerName parameter, type:

Get-Help * -Parameter ComputerName
or
Get-Command -ParameterName ComputerName

To determine whether the ComputerName parameter of a particular cmdlet requires Windows PowerShell remoting, see the parameter description. To display the parameter description, type:

Get-Help <cmdlet-name> -Parameter ComputerName

For example:

Get-Help Get-Process -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.

```
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 Windows PowerShell prompt, type:

```
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:

```
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 Windows PowerShell session (PSSession) that is connected to the remote computer.

When you use the ComputerName parameter of Invoke-Command or Enter-PSSession, Windows PowerShell establishes a temporary connection. Windows 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, Windows 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, Windows PowerShell runs the commands on all of the specified computers or in all of the specified PSSessions.

Windows 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?

Windows 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.

```
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 Windows 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, Windows 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 parameters 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 Windows 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, Windows 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 Windows 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 Windows PowerShell background job is a Windows 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 Windows PowerShell, see about_Jobs and about_Remote_Jobs.

CAN I RUN WINDOWS PROGRAMS ON A REMOTE COMPUTER?

You can use Windows PowerShell remote commands to run Windows-based programs on remote computers. For example, you can run Shutdown.exe or Ipconfig on a remote computer.

However, you cannot use Windows 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 Windows 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 program on a remote computer, the command prompt does not return until IpConfig 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 Windows 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 Windows 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-SessionConfigurationFile 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:

Get-Command *PSSessionConfiguration

WHAT ARE FAN-IN AND FAN OUT CONFIGURATIONS?

The most common Windows PowerShell remoting scenario involving multiple computers is the one-to-many configuration, in which one local computer (the administrator's computer) runs Windows 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 Windows PowerShell, such as a file server or a kiosk. This is known as the "fan-in" configuration.

Windows PowerShell remoting supports both fan-out and fan-in configurations.

For the fan-out configuration, Windows 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 Windows PowerShell to start the Windows 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, Windows PowerShell uses Internet Information Services (IIS) to host WS-Management, to load the Windows PowerShell plug-in, and to start Windows PowerShell. In this scenario, instead of starting each Windows PowerShell session in a separate process, all Windows 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 Windows 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. Windows 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 XP with SP2:

Use Local Security Settings (Secpol.msc) to change the setting of the "Network Access: Sharing and security model for local accounts" policy in Security Settings\Local Policies\Security Options to "Classic".

-- 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 Windows PowerShell command to add this entry:

```
New-ItemProperty `
–Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System `
–Name LocalAccountTokenFilterPolicy –propertyType DWord –Value 1
```

-- 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 Windows PowerShell prompt.

```
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 Windows PowerShell prompt:

```
Set-Item WSMan:\localhost\Client\TrustedHosts -Value Server01
```

SEE ALSO

about_Remote
about_Profiles
about_PSSessions
about_Remote_Jobs
about_Remote_Variables
Invoke-Command
New-PSSession

TOPIC

about_Remote_Jobs

SHORT DESCRIPTION

Describes how to run background jobs on remote computers.

DETAILED DESCRIPTION

A background job is a command that runs asynchronously without interacting with the current session. The command prompt returns immediately, and you can continue to use the session while the job runs.

By default, background jobs run on the local computer. However, you can use several different procedures to run background jobs on remote computers.

This topic explains how to run a background job on a remote computer. For information about how to run background jobs on a local computer, see about_Jobs. For more information about background jobs, see about_Job_Details.

REMOTE BACKGROUND JOBS

You can run background jobs on remote computers by using three different methods.

-- Start an interactive session with a remote computer, and start a job in the interactive session. The procedures are the same as running a local job, although all actions are performed on the remote computer.

-- Run a background job on a remote computer that returns its results to the local computer. Use this method when you want to collect the results of background jobs and maintain them in a central location on the local computer.

-- Run a background job on a remote computer that maintains its results on the remote computer. Use this method when the job data is more securely maintained on the originating computer.

START A BACKGROUND JOB IN AN INTERACTIVE SESSION

You can start an interactive session with a remote computer and then start a background job during the interactive session. For more information about interactive sessions, see about_Remote, and

see Enter-PSSession.

The procedure for starting a background job in an interactive session is almost identical to the procedure for starting a background job on the local computer. However, all of the operations occur on the remote computer, not the local computer.

STEP 1: ENTER-PSSESSION

Use the Enter-PSSession cmdlet to start an interactive session with a remote computer. You can use the ComputerName parameter of Enter-PSSession to establish a temporary connection for the interactive session. Or, you can use the Session parameter to run the interactive session in a Windows PowerShell session (PSSession).

The following command starts an interactive session on the Server01 computer.

```
C:\PS> Enter-PSSession -computername Server01
```

The command prompt changes to show that you are now connected to the Server01 computer.

```
Server01\C:>
```

STEP 2: START-JOB

To start a background job in the session, use the Start-Job cmdlet.

The following command runs a background job that gets the events in the Windows PowerShell event log on the Server01 computer. The Start-Job cmdlet returns an object that represents the job.

This command saves the job object in the $job variable.

```
Server01\C:> $job = start-job -scriptblock {get-eventlog "Windows PowerShell"}
```

While the job runs, you can use the interactive session to run other commands, including other background jobs. However, you must keep the interactive session open until the job is completed. If you end the session, the job is interrupted, and the results are lost.

STEP 3: GET-JOB

To find out if the job is complete, display the value of the $job variable, or use the Get-Job cmdlet to get the job. The following command uses the Get-Job cmdlet to display the job.

```
Server01\C:> get-job $job

SessionId  Name  State     HasMoreData  Location   Command
---------  ----  -----     -----------  --------   -------
1        Job1  Complete   True        localhost  get-eventlog "Windows PowerShell"
```

The Get-Job output shows that job is running on the "localhost" computer because the job was started on and is running on the same computer (in this case, Server01).

STEP 4: RECEIVE-JOB

To get the results of the job, use the Receive-Job cmdlet. You can display the results in the interactive session or save them to a file on the remote computer. The following command gets the results of the job in the $job variable. The command uses the redirection operator (>) to save the results of the job in the PsLog.txt file on the Server01 computer.

```
Server01\C:> receive-job $job > c:\logs\PsLog.txt
```

STEP 5: EXIT-PSSESSION

To end the interactive session, use the Exit-PSSession cmdlet. The command prompt changes to show that you are back in the original session on the local computer.

```
Server01\C:> Exit-PSSession
C:\PS>
```

STEP 6: INVOKE-COMMAND: GET-CONTENT

To view the contents of the PsLog.txt file on the Server01 computer at any time, start another interactive session, or run a remote command. This type of command is best run in a PSSession (a persistent connection) in case you want to use several commands to investigate and manage the data in the PsLog.txt file. For more information about PSSessions, see about_PSSessions.

The following commands use the New-PSSession cmdlet to create a PSSession that is connected to the Server01 computer, and they use the Invoke-Command cmdlet to run a Get-Content command in the PSSession to view the contents of the file.

```
C:\PS> $s = new-pssession -computername Server01
C:\PS> invoke-command -session $s -scriptblock {get-content c:\logs\pslog.txt}
```

START A REMOTE JOB THAT RETURNS THE RESULTS TO THE LOCAL COMPUTER (ASJOB)

To start a background job on a remote computer that returns the command results to the local computer, use the AsJob parameter of a cmdlet such as the Invoke-Command cmdlet.

When you use the AsJob parameter, the job object is actually created on the local computer even though the job runs on the remote computer. When the job is completed, the results are returned to the local computer.

You can use the cmdlets that contain the Job noun (the Job cmdlets) to manage any job created by any cmdlet. Many of the cmdlets that have AsJob parameters do not use Windows PowerShell remoting, so you can use them even on computers that are not configured for remoting and that do not meet the requirements for remoting.

STEP 1: INVOKE-COMMAND -ASJOB

The following command uses the AsJob parameter of Invoke-Command to start a background job on the Server01 computer. The job runs a Get-Eventlog command that gets the events in the System log. You can use the JobName parameter to assign a display name to the job.

```
invoke-command -computername Server01 -scriptblock {get-eventlog system} -asjob
```

The results of the command resemble the following sample output.

```
SessionId   Name    State     HasMoreData    Location   Command
---------   ----    -----     -----------    --------   -------
1           Job1    Running   True           Server01   get-eventlog system
```

When the AsJob parameter is used, Invoke-Command returns the same type of job object that Start-Job returns. You can save the job object in a variable, or you can use a Get-Job command to get the job.

Note that the value of the Location property shows that the job ran on the Server01 computer.

STEP 2: GET-JOB

To manage a job started by using the AsJob parameter of the Invoke-Command cmdlet, use the Job cmdlets. Because the job object that represents the remote job is on the local computer, you do not need to run remote commands to manage the job.

To determine whether the job is complete, use a Get-Job command. The following command gets all of the jobs that were started in the current session.

```
get-job
```

Because the remote job was started in the current session, a local Get-Job command gets the job. The State property of the job object shows that the command was completed successfully.

```
SessionId   Name   State      HasMoreData     Location   Command
---------   ----   -----      -----------     --------   -------
1           Job1   Completed  True            Server01   get-eventlog system
```

STEP 3: RECEIVE-JOB

To get the results of the job, use the Receive-Job cmdlet. Because the job results are automatically returned to the computer where the job object resides, you can get the results with a local Receive-Job command.

The following command uses the Receive-Job cmdlet to get the results of the job. It uses the session ID to identify the job. This command saves the job results in the $results variable. You can also redirect the results to a file.

```
$results = receive-job -id 1
```

START A REMOTE JOB THAT KEEPS THE RESULTS ON THE REMOTE COMPUTER

To start a background job on a remote computer that keeps the command results on the remote computer, use the Invoke-Command cmdlet to run a Start-Job command on a remote computer. You can use this method to run

background jobs on multiple computers.

When you run a Start-Job command remotely, the job object is created on the remote computer, and the job results are maintained on the remote computer. From the perspective of the job, all operations are local. You are just running commands remotely to manage a local job on the remote computer.

STEP 1: INVOKE-COMMAND START-JOB

Use the Invoke-Command cmdlet to run a Start-Job command on a remote computer.

This command requires a PSSession (a persistent connection). If you use the ComputerName parameter of Invoke-Command to establish a temporary connection, the Invoke-Command command is considered to be complete when the job object is returned. As a result, the temporary connection is closed, and the job is canceled.

The following command uses the New-PSSession cmdlet to create a PSSession that is connected to the Server01 computer. The command saves the PSSession in the $s variable.

```
$s = new-pssession -computername Server01
```

The next command uses the Invoke-Command cmdlet to run a Start-Job command in the PSSession. The Start-Job command and the Get-Eventlog command are enclosed in braces.

```
invoke-command -session $s -scriptblock {start-job -scriptblock {get-eventlog system}}
```

The results resemble the following sample output.

```
Id     Name    State     HasMoreData     Location   Command
--     ----    -----     -----------     --------   -------
2      Job2    Running   True            Localhost  get-eventlog system
```

When you run a Start-Job command remotely, Invoke-Command returns the same type of job object that Start-Job returns. You can save the job object in a variable, or you can use a Get-Job command to get the job.

Note that the value of the Location property shows that the job ran on the local computer, known as "LocalHost", even though the job ran on the Server01 computer. Because the job object is created on the Server01 computer and the job runs on the same computer, it is considered to

be a local background job.

STEP 2: INVOKE-COMMAND GET-JOB

To manage a remote background job, use the Job cmdlets. Because the job object is on the remote computer, you need to run remote commands to get, stop, wait for, or retrieve the job results.

To see if the job is complete, use an Invoke-Command command to run a Get-Job command in the PSSession that is connected to the Server01 computer.

```
invoke-command -session $s -scriptblock {get-job}
```

The command returns a job object. The State property of the job object shows that the command was completed successfully.

```
SessionId      Name    State      HasMoreData     Location   Command
---------      ----    -----      -----------     --------   -------
2              Job2    Completed  True            LocalHost  get-eventlog system
```

STEP 3: INVOKE-COMMAND RECEIVE-JOB

To get the results of the job, use the Invoke-Command cmdlet to run a Receive-Job command in the PSSession that is connected to the Server01 computer.

The following command uses the Receive-Job cmdlet to get the results of the job. It uses the session ID to identify the job. This command saves the job results in the $results variable. It uses the Keep parameter of Receive-Job to keep the result in the job cache on the remote computer.

```
$results = invoke-command -session $s -scriptblock {receive-job -sessionid 2 -keep}
```

You can also redirect the results to a file on the local or remote computer. The following command uses a redirection operator to save the results in a file on the Server01 computer.

```
invoke-command -session $s -command {receive-job -sessionid 2 > c:\logs\pslog.txt}
```

SEE ALSO

about_Jobs
about_Job_Details

about_Remote
about_Remote_Variables
Invoke-Command
Start-Job
Get-Job
Wait-Job
Stop-Job
Remove-Job
New-PSSession
Enter-PSSession
Exit-PSSession

TOPIC

about_Remote_Output

SHORT DESCRIPTION

Describes how to interpret and format the output of remote commands.

LONG DESCRIPTION

The output of a command that was run on a remote computer might look like output of the same command run on a local computer, but there are some significant differences.

This topic explains how to interpret, format, and display the output of commands that are run on remote computers.

DISPLAYING THE COMPUTER NAME

When you use the Invoke-Command cmdlet to run a command on a remote computer, the command returns an object that includes the name of the computer that generated the data. The remote computer name is stored in the PSComputerName property.

For many commands, the PSComputerName is displayed by default. For

example, the following command runs a Get-Culture command on two remote computers, Server01 and Server02. The output, which appears below, includes the names of the remote computers on which the command ran.

```
C:\PS> invoke-command -script {get-culture} -comp Server01, Server02

LCID  Name    DisplayName              PSComputerName
----  ----    -----------              --------------
1033  en-US   English (United States)    Server01
1033  es-AR   Spanish (Argentina)       Server02
```

You can use the HideComputerName parameter of Invoke-Command to hide the PSComputerName property. This parameter is designed for commands that collect data from only one remote computer.

The following command runs a Get-Culture command on the Server01 remote computer. It uses the HideComputerName parameter to hide the PSComputerName property and related properties.

```
C:\PS> invoke-command -scr {get-culture} -comp Server01 -HideComputerName

LCID          Name          DisplayName
----          ----          -----------
1033          en-US         English (United States)
```

You can also display the PSComputerName property if it is not displayed by default.

For example, the following commands use the Format-Table cmdlet to add the PSComputerName property to the output of a remote Get-Date command.

```
C:\PS> $dates = invoke-command -script {get-date} -computername Server01, Server02
C:\PS> $dates | format-table DateTime, PSComputerName -auto

DateTime                      PSComputerName
--------                      --------------
Monday, July 21, 2008 7:16:58 PM    Server01
Monday, July 21, 2008 7:16:58 PM    Server02
```

DISPLAYING THE MACHINENAME PROPERTY

Several cmdlets, including Get-Process, Get-Service, and Get-EventLog, have a ComputerName parameter that gets the objects on a remote computer. These cmdlets do not use Windows PowerShell remoting, so you can use them even on computers that are not configured for remoting in Windows PowerShell.

The objects that these cmdlets return store the name of the remote computer in the MachineName property. (These objects do not have a PSComputerName property.)

For example, this command gets the PowerShell process on the Server01 and Server02 remote computers. The default display does not include the MachineName property.

```
C:\PS> get-process PowerShell -computername server01, server02


Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    920     38    97524     114504   575     9.66   2648 PowerShell
    194      6    24256      32384   142          3020 PowerShell
    352     27    63472      63520   577     3.84   4796 PowerShell
```

You can use the Format-Table cmdlet to display the MachineName property of the process objects.

For example, the following command saves the processes in the $p variable and then uses a pipeline operator (|) to send the processes in $p to the Format-Table command. The command uses the Property parameter of Format-Table to include the MachineName property in the display.

```
C:\PS> $p = get-process PowerShell -comp Server01, Server02
C:\PS> $P | format-table -property ID, ProcessName, MachineName -auto


Id ProcessName MachineName
-- ----------- -----------
2648 PowerShell  Server02
3020 PowerShell  Server01
4796 PowerShell  Server02
```

The following more complex command adds the MachineName property to the default process display. It uses hash tables to specify calculated properties. Fortunately, you do not have to understand it to use it.

(Note that the backtick [`] is the continuation character.)

```
C:\PS> $p = get-process PowerShell -comp Server01, Server02

C:\PS> $p | format-table -property Handles, `
        @{Label="NPM(K)";Expression={[int]($_.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


Handles NPM(K) PM(K)  WS(K) VM(M) CPU(s)   Id ProcessName MachineName
------- ------ -----  ----- ----- ------   -- ----------- -----------
    920     38 97560 114532   576       2648 PowerShell  Server02
    192      6 24132  32028   140       3020 PowerShell  Server01
    438     26 48436  59132   565       4796 PowerShell  Server02
```

DESERIALIZED OBJECTS

When you run remote commands that generate output, the command output is transmitted across the network back to the local computer.

Because most live Microsoft .NET Framework objects (such as the objects that Windows PowerShell cmdlets return) cannot be transmitted over the network, the live objects are "serialized". In other words, the live objects are converted into XML representations of the object and its properties. Then, the XML-based serialized object is transmitted across the network.

On the local computer, Windows PowerShell receives the XML-based serialized object and "deserializes" it by converting the XML-based object into a standard .NET Framework object.

However, the deserialized object is not a live object. It is a snapshot of the object at the time that it was serialized, and it includes properties but no methods. You can use and manage these objects in Windows PowerShell, including passing them in pipelines, displaying selected properties, and formatting them.

Most deserialized objects are automatically formatted for display by entries in the Types.ps1xml or Format.ps1xml files. However, the local computer might not have formatting files for all of the deserialized objects that were generated on a remote computer. When objects are not formatted, all of the properties of each object appear in the console

in a streaming list.

When objects are not formatted automatically, you can use the formatting cmdlets, such as Format-Table or Format-List, to format and display selected properties. Or, you can use the Out-GridView cmdlet to display the objects in a table.

Also, if you run a command on a remote computer that uses cmdlets that you do not have on your local computer, the objects that the command returns might not be formatted properly because you do not have the formatting files for those objects on your computer. To get formatting data from another computer, use the Get-FormatData and Export-FormatData cmdlets.

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.

ORDERING THE RESULTS

The order of the computer names in the ComputerName parameter of cmdlets determines the order in which Windows PowerShell connects to the remote computers. However, the results appear in the order in which the local computer receives them, which might be a different order.

To change the order of the results, use the Sort-Object cmdlet. You can sort on the PSComputerName or MachineName property. You can also sort on another property of the object so that the results from different computers are interspersed.

SEE ALSO

about_Remote
about_Remote_Variables
Format-Table
Get-EventLog
Get-Process
Get-Service
Get-WmiObject
Invoke-Command
Out-GridView
Select-Object

TOPIC

about_Remote_Requirements

SHORT DESCRIPTION

Describes the system requirements and configuration requirements for running remote commands in Windows PowerShell.

LONG DESCRIPTION

This topic describes the system requirements, user requirements, and resource requirements for establishing remote connections and running remote commands in Windows PowerShell. It also provides instructions for configuring remote operations.

Note: Many cmdlets (including the Get-Service, Get-Process, Get-WMIObject, Get-EventLog, and Get-WinEvent cmdlets) get objects from remote computers by using Microsoft .NET Framework methods to retrieve the objects. They do not use the Windows PowerShell remoting infrastructure. The requirements in this document do not apply to these cmdlets.

To find the cmdlets that have a ComputerName parameter but do not use Windows PowerShell remoting, read the description of the ComputerName parameter of the cmdlets.

SYSTEM REQUIREMENTS

To run remote sessions on Windows PowerShell 3.0, the local and remote computers must have the following:

-- Windows PowerShell 3.0 or later
-- The Microsoft .NET Framework 4 or later
-- Windows Remote Management 3.0

To run remote sessions on Windows PowerShell 2.0, the local and remote computers must have the following:

-- Windows PowerShell 2.0 or later
-- The Microsoft .NET Framework 2.0 or later

-- Windows Remote Management 2.0

You can create remote sessions between computers running Windows PowerShell 2.0 and Windows PowerShell 3.0. However, features that run only on Windows PowerShell 3.0, such as the ability to disconnect and reconnect to sessions, are available only when both computers are running Windows PowerShell 3.0.

To find the version number of an installed version of Windows PowerShell, use the $PSVersionTable automatic variable.

Windows Remote Management (WinRM) 3.0 and Microsoft .NET Framework 4 are included in Windows 8, Windows Server 2012, and newer releases of the Windows operating system. WinRM 3.0 is included in Windows Management Framework 3.0 for older operating systems. If the computer does not have the required version of WinRM or the Microsoft .NET Framework, the installation fails.

USER PERMISSIONS

To create remote sessions and run remote commands, by default, the current user must be a member of the Administrators group on the remote computer or provide the credentials of an administrator. Otherwise, the command fails.

The permissions required to create sessions and run commands on a remote computer (or in a remote session on the local computer) are established by the session configuration (also known as an "endpoint") on the remote computer to which the session connects. Specifically, the security descriptor on the session configuration determines who has access to the session configuration and who can use it to connect.

The security descriptors on the default session configurations, Microsoft.PowerShell, Microsoft.PowerShell32, and Microsoft.PowerShell.Workflow, allow access only to members of the Administrators group.

If the current user doesn't have permission to use the session configuration, the command to run a command (which uses a temporary session) or create a persistent session on the remote computer fails. The user can use the ConfigurationName parameter of cmdlets that create sessions to select a different session configuration, if one is available.

Members of the Administrators group on a computer can determine who has permission to connect to the computer remotely by changing the security descriptors on the default session configurations and by creating new session configurations with different security descriptors.

For more informations about session configurations, see

about_Session_Configurations (http://go.microsoft.com/fwlink/?LinkID=145152).

WINDOWS NETWORK LOCATIONS

Beginning in Windows PowerShell 3.0, the Enable-PSRemoting cmdlet can enable remoting on client and server versions of Windows on private, domain, and public networks.

On server versions of Windows with private and domain networks, the Enable-PSRemoting cmdlet creates firewall rules that allow unrestricted remote access. It also creates a firewall rule for public networks that allows remote access only from computers in the same local subnet. This local subnet firewall rule is enabled by default on server versions of Windows on public networks, but Enable-PSRemoting reapplies the rule in case it was changed or deleted.

On client versions of Windows with private and domain networks, by default, the Enable-PSRemoting cmdlet creates firewall rules that allow unrestricted remote access.

To enable remoting on client versions of Windows with public networks, use the SkipNetworkProfileCheck parameter of the Enable-PSRemoting cmdlet. It creates a firewall rule that that allows remote access only from computers in the same local subnet.

To remove the local subnet restriction on public networks and allow remote access from all locations on client and server versions of Windows, use the Set-NetFirewallRule cmdlet in the NetSecurity module. Run the following command:

Set-NetFirewallRule –Name "WINRM-HTTP-In-TCP-PUBLIC" –RemoteAddress Any

In Windows PowerShell 2.0, on server versions of Windows, Enable-PSRemoting creates firewall rules that permit remote access on all networks.

In Windows PowerShell 2.0, on client versions of Windows, Enable-PSRemoting creates firewall rules only on private and domain networks. If the network location is public, Enable-PSRemoting fails.

.

RUN AS ADMINISTRATOR

Administrator privileges are required for the following remoting operations:

-- Establishing a remote connection to the local computer. This is

commonly known as a "loopback" scenario.

-- Managing session configurations on the local computer.

-- Viewing and changing WS-Management settings on the local computer. These are the settings in the LocalHost node of the WSMAN: drive.

To perform these tasks, you must start Windows PowerShell with the "Run as administrator" option even if you are a member of the Administrators group on the local computer.

In Windows 7 and in Windows Server 2008 R2, to start Windows PowerShell with the "Run as administrator" option:

1. Click Start, click All Programs, click Accessories, and then click the Windows PowerShell folder.

2. Right-click Windows PowerShell, and then click "Run as administrator".

To start Windows PowerShell with the "Run as administrator" option:

1. Click Start, click All Programs, and then click the Windows PowerShell folder.

2. Right-click Windows PowerShell, and then click "Run as administrator".

The "Run as administrator" option is also available in other Windows Explorer entries for Windows PowerShell, including shortcuts. Just right-click the item, and then click "Run as administrator".

When you start Windows PowerShell from another program such as Cmd.exe, use the "Run as administrator" option to start the program.

HOW TO CONFIGURE YOUR COMPUTER FOR REMOTING

Computers running all supported versions of Windows can establish remote connections to and run remote commands in Windows PowerShell without any configuration. However, to receive connections, and allow users to create local and remote user-managed Windows PowerShell sessions ("PSSessions") and run commands on the local computer, you must enable Windows PowerShell remoting on the computer.

Windows Server 2012 and newer releases of Windows Server are enabled for Windows PowerShell remoting by default. If the settings are changed,

you can restore the default settings by running the Enable-PSRemoting cmdlet.

On all other supported versions of Windows, you need to run the Enable-PSRemoting cmdlet to enable Windows PowerShell remoting.

The remoting features of Windows PowerShell are supported by the WinRM service, which is the Microsoft implementation of the Web Services for Management (WS-Management) protocol. When you enable Windows PowerShell remoting, you change the default configuration of WS-Management and add system configuration that allow users to connect to WS-Management.

To configure Windows PowerShell to receive remote commands:

1. Start Windows PowerShell with the "Run as administrator" option.

2. At the command prompt, type:
   Enable-PSRemoting

To verify that remoting is configured correctly, run a test command such as the following command, which creates a remote session on the local computer.

```
New-PSSession
```

If remoting is configured correctly, the command will create a session on the local computer and return an object that represents the session. The output should resemble the following sample output:

```
C:\PS> new-pssession

Id Name       ComputerName    State    ConfigurationName
-- ----       ------------    -----    -----
1  Session1    localhost       Opened   Microsoft.PowerShell
```

If the command fails, for assistance, see about_Remote_Troubleshooting.

UNDERSTAND POLICIES

When you work remotely, you use two instances of Windows PowerShell, one on the local computer and one on the remote computer. As a result, your work is affected by the Windows policies and the Windows PowerShell policies on the local and remote computers.

In general, before you connect and as you are establishing the connection,

the policies on the local computer are in effect. When you are using the connection, the policies on the remote computer are in effect.

SEE ALSO

about_Remote
about_Remote_Variables
about_PSSessions
Invoke-Command
Enter-PSSession
New-PSSession

TOPIC

about_Remote_Troubleshooting

SHORT DESCRIPTION

Describes how to troubleshoot remote operations in Windows PowerShell.

LONG DESCRIPTION

This section describes some of the problems that you might encounter when using the remoting features of Windows PowerShell that are based on WS-Management technology and it suggests solutions to these problems.

Before using Windows PowerShell remoting, see about_Remote and about_Remote_Requirements for guidance on configuration and basic use Also, the Help topics for each of the remoting cmdlets, particularly the parameter descriptions, have useful information that is designed to help you avoid problems.

Updated versions of this topic, and other Windows PowerShell help topics, can be downloaded by using the Update-Help cmdlet and can be found online in the Microsoft TechNet Library at http://technet.microsoft.com/library/hh847850(v=wps.630).aspx.

NOTE: To view or change settings for the local computer in the WSMan: drive,

including changes to the session configurations, trusted hosts, ports, or
listeners, start Windows PowerShell with the "Run as administrator" option.

TROUBLESHOOTING PERMISSION AND AUTHENTICATION ISSUES

This section discusses remoting problems that are related to user and
computer permissions and remoting requirements.

HOW TO RUN AS ADMINISTRATOR

ERROR: Access is denied. You need to run this cmdlet from an elevated
process.

To start a remote session on the local computer, or to view or change
settings for the local computer in the WSMan: drive, including changes
to the session configurations, trusted hosts, ports, or listeners,
start Windows PowerShell with the "Run as administrator" option.

To start Windows PowerShell with the "Run as administrator option:

-- Right-click a Windows PowerShell (or Windows PowerShell ISE) icon
and then click "Run as administrator.

To start Windows PowerShell with the "Run as administrator option in
Windows 7 and Windows Server 2008 R2.

-- In the Windows taskbar, right-click the Windows PowerShell icon,
and then click "Run as Administrator."

Note: In Windows Server 2008 R2, the Windows PowerShell icon is pinned
to the taskbar by default.

HOW TO ENABLE REMOTING

ERROR: ACCESS IS DENIED
- or -
ERROR: The connection to the remote host was refused. Verify that the
WS-Management service is running on the remote host and configured to
listen for requests on the correct port and HTTP URL.

No configuration is required to enable a computer to send remote
commands. However, to receive remote commands, Windows PowerShell remoting

must be enabled on the computer. Enabling includes starting the WinRM service, setting the startup type for the WinRM service to Automatic, creating listeners for HTTP and HTTPS connections, and creating default session configurations.

Windows PowerShell remoting is enabled on Windows Server 2012 and newer releases of Windows Server by default. On all other systems, run the Enable-PSRemoting cmdlet to enable remoting. You can also run the Enable-PSRemoting cmdlet to re-enable remoting on Windows Server 2012 and newer releases of Windows Server if remoting is disabled.

To configure a computer to receive remote commands, use the Enable-PSRemoting cmdlet. The following command enables all required remote settings, enables the session configurations, and restarts the WinRM service to make the changes effective.

```
Enable-PSRemoting
```

To suppress all user prompts, type:

```
Enable-PSRemoting -Force
```

For more information, see Enable-PSRemoting.

HOW TO ENABLE REMOTING IN AN ENTERPRISE

ERROR: ACCESS IS DENIED
- or -
ERROR: The connection to the remote host was refused. Verify that the WS-Management service is running on the remote host and configured to listen for requests on the correct port and HTTP URL.

To enable a single computer to receive remote Windows PowerShell commands and accept connections, use the Enable-PSRemoting cmdlet.

To enable remoting for multiple computers in an enterprise, you can use the following scaled options.

-- To configure listeners for remoting, enable the "Allow automatic configuration of listeners" group policy. For instructions, see "How to Enable Listeners by Using a Group Policy" (below).

-- To set the startup type of the Windows Remote Management (WinRM) to Automatic on multiple computers, use the Set-Service cmdlet. For instructions, see "How to Set the Startup Type of the WinrM Service" (below).

-- To enable a firewall exception, use the "Windows Firewall: Allow Local Port Exceptions" group policy. For instructions, see "How to Create a Firewall Exception by Using a Group Policy" (below).

HOW TO ENABLE LISTENERS BY USING A GROUP POLICY

ERROR: ACCESS IS DENIED
- or -
ERROR: The connection to the remote host was refused. Verify that the WS-Management service is running on the remote host and configured to listen for requests on the correct port and HTTP URL.

To configure the listeners for all computers in a domain, enable the "Allow automatic configuration of listeners" policy in the following Group Policy path:

Computer Configuration\Administrative Templates\Windows Components \Windows Remote Management (WinRM)\WinRM service

Enable the policy and specify the IPv4 and IPv6 filters. Wildcards (*) are permitted.

HOW TO ENABLE REMOTING ON PUBLIC NETWORKS

ERROR: Unable to check the status of the firewall

The Enable-PSRemoting cmdlet returns this error when the local network is public and the SkipNetworkProfileCheck parameter is not used in the command.

On server versions of Windows, Enable-PSRemoting succeeds on all network location types. It creates firewall rules that allow remote access to private and domain ("Home" and "Work") networks. For public networks, it creates firewall rules that allows remote access from the same local subnet.

On client versions of Windows, Enable-PSRemoting succeeds on private and domain networks. By default, it fails on public networks, but if you use the SkipNetworkProfileCheck parameter, Enable-PSRemoting succeeds and creates a firewall rule that allows traffic from the same local subnet.

To remove the local subnet restriction on public networks and allow remote access from any location, run the following command:

Set-NetFirewallRule –Name "WINRM-HTTP-In-TCP-PUBLIC" –RemoteAddress Any

The Set-NetFirewallRule cmdlet is exported by the NetSecurity module.

NOTE: In Windows PowerShell 2.0, on computers running server versions of Windows, Enable-PSRemoting creates firewall rules that allow remote access on private, domain and public networks. On computers running client versions of Windows, Enable-PSRemoting creates firewall rules that allow remote access only on private and domain networks.

HOW TO ENABLE A FIREWALL EXCEPTION BY USING A GROUP POLICY

ERROR: ACCESS IS DENIED
- or -
ERROR: The connection to the remote host was refused. Verify that the WS-Management service is running on the remote host and configured to listen for requests on the correct port and HTTP URL.

To enable a firewall exception for in all computers in a domain, enable the "Windows Firewall: Allow local port exceptions" policy in the following Group Policy path:

Computer Configuration\Administrative Templates\Network
\Network Connections\Windows Firewall\Domain Profile

This policy allows members of the Administrators group on the computer to use Windows Firewall in Control Panel to create a firewall exception for the Windows Remote Management service.

HOW TO SET THE STARTUP TYPE OF THE WINRM SERVICE

ERROR: ACCESS IS DENIED

Windows PowerShell remoting depends upon the Windows Remote Management (WinRM) service. The service must be running to support remote commands.

On server versions of Windows, the startup type of the Windows Remote Management (WinRM) service is Automatic.

However, on client versions of Windows, the WinRM service is disabled by default.

To set the startup type of a service on a remote computer, use the Set-Service cmdlet.

To run the command on multiple computers, you can create a text file or CSV file of the computer names.

For example, the following commands get a list of computer names from the Servers.txt file and then sets the startup type of the WinRM service on all of the computers to Automatic.

```
C:\PS> $servers = Get-Content servers.txt

C:\PS> Set-Service WinRM -ComputerName $servers -startuptype Automatic
```

To see the results use the Get-WMIObject cmdlet with the Win32_Service object. For more information, see Set-Service.

HOW TO RECREATE THE DEFAULT SESSION CONFIGURATIONS

ERROR: ACCESS IS DENIED

To connect to the local computer and run commands remotely, the local computer must include session configurations for remote commands.

When you use Enable-PSRemoting, it creates default session configurations on the local computer. Remote users use these session configurations whenever a remote command does not include the ConfigurationName parameter.

If the default configurations on a computer are unregistered or deleted, use the Enable-PSRemoting cmdlet to recreate them. You can use this cmdlet repeatedly. It does not generate errors if a feature is already configured.

If you change the default session configurations and want to restore the original default session configurations, use the Unregister-PSSessionConfiguration cmdlet to delete the changed session configurations and then use the Enable-PSRemoting cmdlet to restore them. Enable-PSRemoting does not change existing session configurations.

Note: When Enable-PSRemoting restores the default session configuration, it does not create explicit security descriptors for the configurations. Instead, the configurations inherit the security descriptor of the RootSDDL, which is secure by default.

To see the RootSDDL security descriptor, type:

```
Get-Item wsman:\localhost\Service\RootSDDL
```

To change the RootSDDL, use the Set-Item cmdlet in the WSMan: drive. To change the security descriptor of a session configuration, use the

Set-PSSessionConfiguration cmdlet with the SecurityDescriptorSDDL or ShowSecurityDescriptorUI parameters.

For more information about the WSMan: drive, see the Help topic for the WSMan provider ("Get-Help wsman").

HOW TO PROVIDE ADMINISTRATOR CREDENTIALS

ERROR: ACCESS IS DENIED

To create a PSSession or run commands on a remote computer, by default, the current user must be a member of the Administrators group on the remote computer. Credentials are sometimes required even when the current user is logged on to an account that is a member of the Administrators group.

If the current user is a member of the Administrators group on the remote computer, or can provide the credentials of a member of the Administrators group, use the Credential parameter of the New-PSSession, Enter-PSSession or Invoke-Command cmdlets to connect remotely.

For example, the following command provides the credentials of an Administrator.

```
Invoke-Command -ComputerName Server01 -Credential Domain01\Admin01
```

For more information about the Credential parameter, see New-PSSession, Enter-PSSession or Invoke-Command.

HOW TO ENABLE REMOTING FOR NON-ADMINISTRATIVE USERS

ERROR: ACCESS IS DENIED

To establish a PSSession or run a command on a remote computer, the user must have permission to use the session configurations on the remote computer.

By default, only members of the Administrators group on a computer have permission to use the default session configurations. Therefore, only members of the Administrators group can connect to the computer remotely.

To allow other users to connect to the local computer, give the user Execute permissions to the default session configurations on the local computer.

The following command opens a property sheet that lets you change the security descriptor of the default Microsoft.PowerShell session configuration on the local computer.

```
Set-PSSessionConfiguration Microsoft.PowerShell -ShowSecurityDescriptorUI
```

For more information, see about_Session_Configurations.

HOW TO ENABLE REMOTING FOR ADMINISTRATORS IN OTHER DOMAINS

ERROR: ACCESS IS DENIED

When a user in another domain is a member of the Administrators group on the local computer, the user cannot connect to the local computer remotely with Administrator privileges. By default, remote connections from other domains run with only standard user privilege tokens.

However, you can use the LocalAccountTokenFilterPolicy registry entry to change the default behavior and allow remote users who are members of the Administrators group to run with Administrator privileges.

Caution: The LocalAccountTokenFilterPolicy entry disables user account control (UAC) remote restrictions for all users of all affected computers. Consider the implications of this setting carefully before changing the policy.

To change the policy, use the following command to set the value of the LocalAccountTokenFilterPolicy registry entry to 1.

```
C:\PS> New-ItemProperty -Name LocalAccountTokenFilterPolicy -Path `
  HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System -PropertyType `
  DWord -Value 1
```

HOW TO USE AN IP ADDRESS IN A REMOTE COMMAND

ERROR: The WinRM client cannot process the request. If the authentication scheme is different from Kerberos, or if the client computer is not joined to a domain, then HTTPS transport must be used or the destination machine must be added to the TrustedHosts configuration setting.

The ComputerName parameters of the New-PSSession, Enter-PSSession and Invoke-Command cmdlets accept an IP address as a valid value. However,

because Kerberos authentication does not support IP addresses, NTLM authentication is used by default whenever you specify an IP address.

When using NTLM authentication, the following procedure is required for remoting.

1. Configure the computer for HTTPS transport or add the IP addresses of the remote computers to the TrustedHosts list on the local computer.

   For instructions, see "How to Add a Computer to the TrustedHosts List" below.

2. Use the Credential parameter in all remote commands.

   This is required even when you are submitting the credentials of the current user.

HOW TO CONNECT REMOTELY FROM A WORKGROUP-BASED COMPUTER

ERROR: The WinRM client cannot process the request. If the authentication scheme is different from Kerberos, or if the client computer is not joined to a domain, then HTTPS transport must be used or the destination machine must be added to the TrustedHosts configuration setting.

When the local computer is not in a domain, the following procedure is required for remoting.

1. Configure the computer for HTTPS transport or add the names of the remote computers to the TrustedHosts list on the local computer.

   For instructions, see "How to Add a Computer to the TrustedHosts List" below.

2. Verify that a password is set on the workgroup-based computer. If a password is not set or the password value is empty, you cannot run remote commands.

   To set password for your user account, use User Accounts in Control Panel.

3. Use the Credential parameter in all remote commands.

This is required even when you are submitting the credentials
of the current user.

HOW TO ADD A COMPUTER TO THE TRUSTED HOSTS LIST

The TrustedHosts item can contain a comma-separated list of computer names, IP addresses, and fully-qualified domain names. Wildcards are permitted.

To view or change the trusted host list, use the WSMan: drive. The TrustedHost item is in the WSMan:\localhost\Client node.

Only members of the Administrators group on the computer have permission to change the list of trusted hosts on the computer.

Caution: The value that you set for the TrustedHosts item affects all users of the computer.

To view the list of trusted hosts, use the following command:

```
Get-Item wsman:\localhost\Client\TrustedHosts
```

You can also use the Set-Location cmdlet (alias = cd) to navigate though the WSMan: drive to the location.
For example: "cd WSMan:\localhost\Client; dir".

To add all computers to the list of trusted hosts, use the following command, which places a value of * (all) in the ComputerName

```
Set-Item wsman:localhost\client\trustedhosts -Value *
```

You can also use a wildcard character (*) to add all computers in a particular domain to the list of trusted hosts. For example, the following command adds all of the computers in the Fabrikam domain to the list of trusted hosts.

```
Set-Item wsman:localhost\client\trustedhosts *.fabrikam.com
```

To add the names of particular computers to the list of trusted hosts, use the following command format:

```
Set-Item wsman:\localhost\Client\TrustedHosts -Value <ComputerName>[,<ComputerName>]
```

where each value <ComputerName> must have the following format:

```
<Computer>.<Domain>.<Company>.<top-level-domain>
```

For example:

```
Set-Item wsman:\localhost\Client\TrustedHosts -Value Server01.Domain01.Fabrikam.com
```

To add a computer name to an existing list of trusted hosts, first save the current value in a variable, and then set the value to a comma-separated list that includes the current and new values.

For example, to add the Server01 computer to an existing list of trusted hosts, use the following command

```
$curValue = (Get-Item wsman:\localhost\Client\TrustedHosts).value

Set-Item wsman:\localhost\Client\TrustedHosts -Value "$curValue,
Server01.Domain01.Fabrikam.com"
```

To add the IP addresses of particular computers to the list of trusted hosts, use the following command format:

```
Set-Item wsman:\localhost\Client\TrustedHosts -Value <IP Address>
```

For example:

```
Set-Item wsman:\localhost\Client\TrustedHosts -Value 172.16.0.0
```

To add a computer to the TrustedHosts list of a remote computer, use the Connect-WSMan cmdlet to add a node for the remote computer to the WSMan: drive on the local computer. Then use a Set-Item command to add the computer.

For more information about the Connect-WSMan cmdlet, see Connect-WSMan.

TROUBLESHOOTING COMPUTER CONFIGURATION ISSUES

This section discusses remoting problems that are related to particular configurations of a computer, domain, or enterprise.

HOW TO CONFIGURE REMOTING ON ALTERNATE PORTS

ERROR: The connection to the specified remote host was refused. Verify that the WS-Management service is running on the remote host and configured to listen for requests on the correct port and HTTP URL.

Windows PowerShell remoting uses port 80 for HTTP transport by default. The default port is used whenever the user does not specify the ConnectionURI or Port parameters in a remote command.

To change the default port that Windows PowerShell uses, use Set-Item cmdlet in the WSMan: drive to change the Port value in the listener leaf node.

For example, the following command changes the default port to 8080.

```
Set-Item wsman:\localhost\listener\listener*\port -Value 8080
```

HOW TO CONFIGURE REMOTING WITH A PROXY SERVER

ERROR: The client cannot connect to the destination specified in the request. Verify that the service on the destination is running and is accepting requests.

Because Windows PowerShell remoting uses the HTTP protocol, it is affected by HTTP proxy settings. In enterprises that have proxy servers, users cannot access a Windows PowerShell remote computer directly.

To resolve this problem, use proxy setting options in your remote command. The following settings are available:

- -- ProxyAccessType
- -- ProxyAuthentication
- -- ProxyCredential

To set these options for a particular command, use the following procedure:

1. Use the ProxyAccessType, ProxyAuthentication, and ProxyCredential parameters of the New-PSSessionOption cmdlet to create a session option object with the proxy settings for your enterprise. Save the

option object is a variable.

2. Use the variable that contains the option object as the value of the SessionOption parameter of a New-PSSession, Enter-PSSession, or Invoke-Command command.

For example, the following command creates a session option object with proxy session options and then uses the object to create a remote session.

```
C:\PS> $SessionOption = New-PSSessionOption -ProxyAccessType IEConfig `
    -ProxyAuthentication Negotiate -ProxyCredential Domain01\User01

C:\PS> New-PSSession -ConnectionURI https://www.fabrikam.com
```

For more information about the New-PSSessionOption cmdlet, see New-PSSessionOption.

To set these options for all remote commands in the current session, use the option object that New-PSSessionOption creates in the value of the $PSSessionOption preference variable. For more information about the $PSSessionOption preference variable, see about_Preference_Variables.

To set these options for all remote commands all Windows PowerShell sessions on the local computer, add the $PSSessionOption preference variable to your Windows PowerShell profile. For more information about Windows PowerShell profiles, see about_Profiles.

HOW TO DETECT A 32-BIT SESSION ON A 64-BIT COMPUTER

```
ERROR: The term "<tool-Name>" 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.
```

If the remote computer is running a 64-bit version of Windows, and the remote command is using a 32-bit session configuration, such as Microsoft.PowerShell32, Windows Remote Management (WinRM) loads a WOW64 process and Windows automatically redirects all references to the %Windir%\System32 directory to the %windir%\SysWOW64 directory.

As a result, if you try to use tools in the System32 directory that do not have counterparts in the SysWow64 directory, such as Defrag.exe, the tools cannot be found in the directory.

To find the processor architecture that is being used in the session, use the value of the PROCESSOR_ARCHITECTURE environment variable. The following command finds the processor architecture of the session in the $s variable.

```
C:\PS> $s = New-PSSession -ComputerName Server01 -configurationName CustomShell

C:\PS> invoke-command -session $s {$env:PROCESSOR_ARCHITECTURE}
x86
```

For more information about session configurations, see about_session_configurations.

TROUBLESHOOTING POLICY AND PREFERENCE ISSUES

This section discusses remoting problems that are related to policies and preferences set on the local and remote computers.

HOW TO CHANGE THE EXECUTION POLICY FOR IMPORT-PSSESSION AND IMPORT-MODULE

ERROR: Import-Module: File <filename> cannot be loaded because the execution of scripts is disabled on this system.

The Import-PSSession and Export-PSSession cmdlets create modules that contains unsigned script files and formatting files.

To import the modules that are created by these cmdlets, either by using Import-PSSession or Import-Module, the execution policy in the current session cannot be Restricted or AllSigned. (For information about Windows PowerShell execution policies, see about_Execution_Policies.

To import the modules without changing the execution policy for the local computer that is set in the registry, use the Scope parameter of Set-ExecutionPolicy to set a less restrictive execution policy for a single process.

For example, the following command starts a process with the RemoteSigned execution policy. The execution policy change affects only the current process and does not change the Windows PowerShell ExecutionPolicy registry setting.

```
Set-ExecutionPolicy -Scope process -ExecutionPolicy RemoteSigned
```

You can also use the ExecutionPolicy parameter of PowerShell.exe to start a single session with a less restrictive execution policy.

PowerShell.exe -ExecutionPolicy RemoteSigned

For more information about the cmdlets, see Import-PSSession, Export-PSSession, and Import-Module. For more information about execution policies, see about_Execution_Policies. For more information about the PowerShell.exe console help options, type "PowerShell.exe -?".

HOW TO SET AND CHANGE QUOTAS

ERROR: The total data received from the remote client exceeded allowed maximum.

You can use quotas to protect the local computer and the remote computer from excessive resource use, both accidental and malicious.

The following quotas are available in the basic configuration.

-- The WSMan provider (WSMan:) provides several quota settings, such as the MaxEnvelopeSizeKB and MaxProviderRequests settings in the WSMan:\<ComputerName> node and the MaxConcurrentOperations, MaxConcurrentOperationsPerUser, and MaxConnections settings in the WSMan:\<ComputerName>\Service node.

-- You can protect the local computer by using the MaximumReceivedDataSizePerCommand and MaximumReceivedObjectSize parameters of the New-PSSessionOption cmdlet and the $PSSessionOption preference variable.

-- You can protect the remote computer by adding restrictions to the session configurations, such as by using the MaximumReceivedDataSizePerCommandMB and MaximumReceivedObjectSizeMB parameters of the Register-PSSessionConfiguration cmdlet.

When quotas conflict with a command, Windows PowerShell generates an error.

To resolve the error, change the remote command to comply with the quota. Or, determine the source of the quota, and then increase the quota to allow the command to complete.

For example, the following command increases the object size quota in the Microsoft.PowerShell session configuration on the remote computer from 10 MB (the default value) to 11 MB.

```
Set-PSSessionConfiguration -Name microsoft.PowerShell `
   -MaximumReceivedObjectSizeMB 11 -Force
```

For more information about the New-PSSessionOption cmdlet, see New-PSSessionOption.

For more information about the WS-Management quotas, see the Help topic for the WSMan provider (type "Get-Help WSMan").

HOW TO RESOLVE TIMEOUT ERRORS

ERROR: The WS-Management service cannot complete the operation within the time specified in OperationTimeout.

You can use timeouts to protect the local computer and the remote computer from excessive resource use, both accidental and malicious. When timeouts are set on both the local and remote computer, Windows PowerShell uses the shortest timeout settings.

The following timeouts are available in the basic configuration.

-- The WSMan provider (WSMan:) provides several client-side and service-side timeout settings, such as the MaxTimeoutms setting in the WSMan:\<ComputerName> node and the EnumerationTimeoutms and MaxPacketRetrievalTimeSeconds settings in the WSMan:\<ComputerName>\Service node.

-- You can protect the local computer by using the CancelTimeout, IdleTimeout, OpenTimeout, and OperationTimeout parameters of the New-PSSessionOption cmdlet and the $PSSessionOption preference variable.

-- You can also protect the remote computer by setting timeout values programmatically in the session configuration for the session.

When a timeout value does not permit a operation to complete, Windows PowerShell terminates the operation and generates an error.

To resolve the error, change the command to complete within the timeout interval or determine the source of the timeout limit and increase the timeout interval to allow the command to complete.

For example, the following commans use the New-PSSessionOption cmdlet to create a session option object with an OperationTimeout value of 4 minutes (in MS) and then use the session option object to create a remote session.

```
C:\PS> $pso = New-PSSessionoption -OperationTimeout 240000

C:\PS> New-PSSession -ComputerName Server01 -sessionOption $pso
```

For more information about the WS-Management timeouts, see the Help topic for the WSMan provider (type "Get-Help WSMan").

For more information about the New-PSSessionOption cmdlet, see New-PSSessionOption.

TROUBLESHOOTING UNRESPONSIVE BEHAVIOR

This section discusses remoting problems that prevent a command from completing and prevent or delay the return of the Windows PowerShell prompt.

HOW TO INTERRUPT A COMMAND

Some native Windows programs, such as programs with a user interface, console applications that prompt for input, and console applications that use the Win32 console API, do not work correctly in the Windows PowerShell remote host.

When you use these programs, you might see unexpected behavior, such as no output, partial output, or a remote command that does not complete.

To end an unresponsive program, type CTRL + C. To view any errors that might have been reported, type "$error" in the local host and the remote session.

HOW TO RECOVER FROM AN OPERATION FAILURE

ERROR: The I/O operation has been aborted because of either a thread exit or an application request.

This error is returned when an operation is terminated before it completes. Typically, it occurs when the WinRM service stops or restarts while other WinRM operations are in progress.

To resolve this issue, verify that the WinRM service is running and try the command again.

1. Start Windows PowerShell with the "Run as administrator" option.
2. Run the following command:

   Start-Service WinRM

3. Re-run the command that generated the error.

SEE ALSO

Online version: http://technet.microsoft.com/library/hh847850(v=wps.630).aspx
about_Remote
about_Remote_Requirements
about_Remote_Variables

TOPIC

about_Remote_Variables

SHORT DESCRIPTION

Explains how to use local and remote variables in remote commands.

LONG DESCRIPTION

You can use variables in commands that you run on remote computers. Simply assign a value to the variable and then use the variable in place of the value.

By default, the variables in remote commands are assumed to be defined in the session in which the command runs. You can also use variables that are defined in the local session, but you must identify them as local variables in the command.

USING REMOTE VARIABLES

Windows PowerShell assumes that the variables used in remote commands are defined in the session in which the command runs.

In the following example, the $ps variable is defined in the temporary session in which the Get-WinEvent command runs.

```
PS C:\>Invoke-Command -ComputerName S1 -ScriptBlock {$ps = "Windows PowerShell"; Get-WinEvent -LogName $ps}
```

Similarly, when the command runs in a persistent session (PSSession), the remote variable must be defined in the same PSSession.

```
PS C:\>$s = New-PSSession -ComputerName S1

PS C:\>Invoke-Command -ComputerName S1 -ScriptBlock {$ps = "Windows PowerShell"}

PS C:\>Invoke-Command -Sessions $s -ScriptBlock {Get-WinEvent -LogName $ps}
```

USING LOCAL VARIABLES

You can also use local variables in remote commands, but you must indicate that the variable is defined in the local session.

Beginning in Windows PowerShell 3.0, you can use the Using scope modifier to identify a local variable in a remote command.

The syntax of Using is as follows:

The syntax is:

```
$Using:<VariableName>
```

In the following example, the $ps variable is created in the local session, but is used in the session in which the command runs. The Using scope modifier identifies $ps as a local variable.

```
PS C:\>$ps = "Windows PowerShell"
PS C:\>Invoke-Command -ComputerName S1 -ScriptBlock {Get-WinEvent -LogName $Using:ps}
```

You can also use the Using scope modifier in PSSessions.

```
PS C:\>$s = New-PSSession -ComputerName S1

PS C:\>$ps = "Windows PowerShell"

PS C:\>Invoke-Command -Sessions $s -ScriptBlock {Get-WinEvent -LogName $Using:ps}
```

USING LOCAL VARIABLES IN WINDOWS POWERSHELL 2.0

You can use local variables in a remote command by defining parameters for the remote command and using the ArgumentList parameter of the Invoke-Command cmdlet to specify the local variable as the parameter value.

This command format is valid on Windows PowerShell 2.0 and later versions of Windows PowerShell.

-- Use the param keyword to define parameters for the remote command. The parameter names are placeholders that do not need to match the name of the local variable.

-- Use the parameters defined by the param keyword in the command.

-- Use the ArgumentList parameter of the Invoke-Command cmdlet to specify the local variable as the parameter value.

For example, the following commands define the $ps variable in the local session and then use it in a remote command. The command uses $log as the parameter name and the local variable, $ps, as its value.

```
C:\PS>$ps = "Windows PowerShell"

C:\PS>Invoke-Command -ComputerName S1 -ScriptBlock {param($log) Get-WinEvent -logname $log}
-ArgumentList $ps
```

KEYWORDS

about_Using

SEE ALSO

about_Remote
about_PSSessions
about_Scopes
Enter-PSSession
Invoke-Command
New-PSSession

TOPIC

about_Requires

SHORT DESCRIPTION

Prevents a script from running without the required elements.

LONG DESCRIPTION

The #Requires statement prevents a script from running unless the Windows PowerShell version, modules, snap-ins, and module and snap-in version prerequisites are met. If the prerequisites are not met, Windows PowerShell does not run the script.

You can use #Requires statements in any script. You cannot use them in functions, cmdlets, or snap-ins.

SYNTAX

```
#Requires -Version <N>[.<n>]
#Requires –PSSnapin <PSSnapin-Name> [-Version <N>[.<n>]]
#Requires -Modules { <Module-Name> | <Hashtable> }
#Requires –ShellId <ShellId>
#Requires -RunAsAdministrator
```

RULES FOR USE

- The #Requires statement must be the first item on a line in a script.
- A script can include more than one #Requires statement.
- The #Requires statements can appear on any line in a script.

PARAMETERS

-Version <N>[.<n>]

Specifies the minimum version of Windows PowerShell that the script requires. Enter a major version number and optional minor version number.

For example:

```
#Requires -Version 3.0
```

-PSSnapin <PSSnapin-Name> [-Version <N>[.<n>]]
Specifies a Windows PowerShell snap-in that the script requires. Enter the snap-in name and an optional version number.

For example:
#Requires –PSSnapin DiskSnapin -Version 1.2

-Modules <Module-Name> | <Hashtable>
Specifies Windows PowerShell modules that the script requires. Enter the module name and an optional version number. The Modules parameter is introduced in Windows PowerShell 3.0.

If the required modules are not in the current session, Windows PowerShell imports them. If the modules cannot be imported, Windows PowerShell throws a terminating error.

For each module, type the module name (<String>) or a hash table with the following keys. The value can be a combination of strings and hash tables.

-- ModuleName. This key is required.
-- ModuleVersion. This key is required.
-- GUID. This key is optional.

For example,
#Requires -Modules PSWorkflow, @{ModuleName="PSScheduledJob";ModuleVersion=1.0.0.0}

-ShellId
Specifies the shell that the script requires. Enter the shell ID.

For example,
#Requires –ShellId MyLocalShell

-RunAsAdministrator
When this switch parameter is added to your requires statement, it specifies that the Windows PowerShell session in which you are running the script must be started with elevated user rights (Run as Administrator).

For example,
#Requires -RunAsAdministrator

EXAMPLES

The following script has two #Requires statements. If the requirements specified in both statements are not met, the script does not run. Each #Requires statement must be the first item on a line:

```
#Requires –Modules PSWorkflow
#Requires –Version 3
Param
(
  [parameter(Mandatory=$true)]
  [String[]]
  $Path
)
...
```

NOTES

In Windows PowerShell 3.0, the Windows PowerShell Core packages appear as modules in sessions started by using the InitialSessionState.CreateDefault2 method, such as sessions started in the Windows PowerShell console. Otherwise, they appear as snap-ins. The exception is Microsoft.PowerShell.Core, which is always a snap-in.

SEE ALSO

about_Automatic_Variables
about_Language_Keywords
about_PSSnapins
get-PSSnapin

TOPIC

about_Reserved_Words

SHORT DESCRIPTION

Lists the reserved words that cannot be used as identifiers because they have a special meaning in Windows PowerShell.

LONG DESCRIPTION

There are certain words that have special meaning in Windows PowerShell. When these words appear without quotation marks, Windows PowerShell attempts to apply their special meaning rather than treating them as character strings. To use these words as parameter arguments in a command or script without invoking their special meaning, enclose the reserved words in quotation marks.

The following are the reserved words in Windows PowerShell:

```
Begin           Exit            Process
Break           Filter          Return
Catch           Finally         Sequence
Class           For             Switch
Continue        ForEach         Throw
Data            From            Trap
Define          Function        Try
Do              If              Until
DynamicParam    In              Using
Else            InlineScript    Var
ElseIf          Parallel        While
End             Param           Workflow
```

For more information about language statements, including Foreach, If, For, and While, type "Get-help", type the prefix "about_", and then type the name of the statement. For example, to get information about the Foreach statement, type:

Get-Help about_ForEach

For information about the Filter statement or the Return statement syntax, type:

Get-Help about_Functions

For information about other reserved words, type:

Get-Help <Reserved_Word>

SEE ALSO

about_Command_Syntax
about_Escape_Characters
about_Language_Keywords
about_Parsing
about_Quoting_Rules
about_Script_Blocks
about_Special_Characters

TOPIC

about_Return

SHORT DESCRIPTION

Exits the current scope, which can be a function, script, or script block.

LONG DESCRIPTION

The Return keyword exits a function, script, or script block. It can be used to exit a scope at a specific point, to return a value, or to indicate that the end of the scope has been reached.

Users who are familiar with languages like C or C# might want to use the Return keyword to make the logic of leaving a scope explicit.

In Windows PowerShell, the results of each statement are returned as output, even without a statement that contains the Return keyword. Languages like C or C# return only the value or values that are specified by the Return keyword.

Syntax

The syntax for the Return keyword is as follows:

```
return [<expression>]
```

The Return keyword can appear alone, or it can be followed by a value or expression, as follows:

```
return
```

```
return $a
return (2 + $a)
```

Examples

The following example uses the Return keyword to exit a function at a specific point if a conditional is met:

```
function ScreenPassword($instance)
{
   if (!($instance.screensaversecure)) {return $instance.name}
   <additional statements>
}

foreach ($a in @(get-wmiobject win32_desktop)) { ScreenPassword($a) }
```

This script checks each user account. The ScreenPassword function returns the name of any user account that does not have a password-protected screen saver. If the screen saver is password protected, the function completes any other statements to be run, and Windows PowerShell does not return any value.

In Windows PowerShell, values can be returned even if the Return keyword is not used. The results of each statement are returned. For example, the following statements return the value of the $a variable:

```
$a
return
```

The following statement also returns the value of $a:

```
return $a
```

The following example includes a statement intended to let the user know that the function is performing a calculation:

```
function calculation {
   param ($value)

   "Please wait. Working on calculation..."
   $value += 73
   return $value
   }
```

Running this function and assigning the result to a variable has the following effect:

```
C:\PS> $a = calculation 14
C:\PS>
```

The "Please wait. Working on calculation..." string is not displayed. Instead, it is assigned to the $a variable, as in the following example:

```
C:\PS> $a
Please wait. Working on calculation...
87
```

Both the informational string and the result of the calculation are returned by the function and assigned to the $a variable.

SEE ALSO

Exit keyword in about_Language_Keywords
about_Functions
about_Scopes
about_Script_Blocks

TOPIC

about_Run_With_PowerShell

SHORT DESCRIPTION

Explains how to use the "Run with PowerShell" feature to run a script from a file system drive.

LONG DESCRIPTION

Beginning in Windows PowerShell 3.0, you can use the "Run with PowerShell" feature to run scripts from File Explorer in Windows 8 and Windows Server 2012 and from Windows Explorer in earlier versions of Windows.

The "Run with PowerShell" feature is designed to run scripts that do not have required parameters and do not return output to the command prompt.

When you use the "Run with PowerShell" feature, the Windows PowerShell console window appears only briefly, if at all. You cannot interact with it.

To use the "Run with PowerShell" feature:

In File Explorer (or Windows Explorer), right-click the script file name and then select "Run with PowerShell".

The "Run with PowerShell" feature starts a Windows PowerShell session that has an execution policy of Bypass, runs the script, and closes the session.

It runs a command that has the following format:

PowerShell.exe -File <FileName> -ExecutionPolicy Bypass

"Run with PowerShell" sets the Bypass execution policy only for the session (the current instance of the PowerShell process) in which the script runs. This feature does not change the execution policy for the computer or the user.

The "Run with PowerShell" feature is affected only by the AllSigned execution policy. If the AllSigned execution policy is effective for the computer or the user, "Run with PowerShell" runs only signed scripts. "Run with PowerShell" is not affected by any other execution policy. For more information, see about_Execution_Policies.

Troubleshooting Note: Run with PowerShell command might prompt you to confirm the execution policy change.

SEE ALSO

about_Execution_Policies
about_Group_Policy_Settings
about_Scripts
"Running Scripts" (http://go.microsoft.com/fwlink/?LinkId=257680)

```
Name                    Category  Module            Synopsis
----                    --------  ------            --------
about_Scheduled_Jobs    HelpFile                    Describes scheduled jobs and explains how to use
and manage
about_Scheduled_Jobs    HelpFile                    Describes scheduled jobs and explains how to use
and manage
```

```
Name                    Category  Module            Synopsis
----                    --------  ------            --------
about_Scheduled_Jobs_Advanced    HelpFile                    Explains advanced scheduled job topics,
including the file structure
about_Scheduled_Jobs_Advanced    HelpFile                    Explains advanced scheduled job topics,
including the file structure
```

```
Name                    Category  Module            Synopsis
----                    --------  ------            --------
about_Scheduled_Jobs_Basics      HelpFile                    Explains how to create and manage scheduled
jobs.
about_Scheduled_Jobs_Basics      HelpFile                    Explains how to create and manage scheduled
jobs.
```

```
Name                    Category  Module            Synopsis
----                    --------  ------            --------
about_Scheduled_Jobs_Troublesh... HelpFile                    Explains how to resolve problems with
scheduled jobs
about_Scheduled_Jobs_Troublesh... HelpFile                    Explains how to resolve problems with
scheduled jobs
```

TOPIC

about_Scopes

SHORT DESCRIPTION

Explains the concept of scope in Windows PowerShell and shows how to set and change the scope of elements.

LONG DESCRIPTION

Windows PowerShell protects access to variables, aliases, functions, and Windows PowerShell drives (PSDrives) by limiting where they can be read and changed. By enforcing a few simple rules for scope, Windows PowerShell helps to ensure that you do not inadvertently change an item that should not be changed.

The following are the basic rules of scope:

- An item you include in a scope is visible in the scope in which it was created and in any child scope, unless you explicitly make it private. You can place variables, aliases, functions, or Windows PowerShell drives in one or more scopes.

- An item that you created within a scope can be changed only in the scope in which it was created, unless you explicitly specify a different scope.

If you create an item in a scope, and the item shares its name with an item in a different scope, the original item might be hidden under the new item. But, it is not overridden or changed.

Windows PowerShell Scopes

Scopes in Windows PowerShell have both names and numbers. The named scopes specify an absolute scope. The numbers are relative and reflect the relationship between scopes.

Global:

The scope that is in effect when Windows PowerShell starts. Variables and functions that are present when Windows PowerShell starts have been created in the global scope. This includes automatic variables and

preference variables. This also includes the variables, aliases, and functions that are in your Windows PowerShell profiles.

Local:
The current scope. The local scope can be the global scope or any other scope.

Script:
The scope that is created while a script file runs. Only the commands in the script run in the script scope. To the commands in a script, the script scope is the local scope.

Private:
Items in private scope cannot be seen outside of the current scope. You can use private scope to create a private version of an item with the same name in another scope.

Numbered Scopes:
You can refer to scopes by name or by a number that describes the relative position of one scope to another. Scope 0 represents the current, or local, scope. Scope 1 indicates the immediate parent scope. Scope 2 indicates the parent of the parent scope, and so on. Numbered scopes are useful if you have created many recursive scopes.

Parent and Child Scopes

You can create a new scope by running a script or function, by creating a session, or by starting a new instance of Windows PowerShell. When you create a new scope, the result is a parent scope (the original scope) and a child scope (the scope that you created).

In Windows PowerShell, all scopes are child scopes of the global scope, but you can create many scopes and many recursive scopes.

Unless you explicitly make the items private, the items in the parent scope are available to the child scope. However, items that you create and change in the child scope do not affect the parent scope, unless you explicitly specify the scope when you create the items.

Inheritance

A child scope does not inherit the variables, aliases, and functions from the parent scope. Unless an item is private, the child scope can view the items in the parent scope. And, it can change the items by explicitly specifying the parent scope, but the items are not part of the child scope.

However, a child scope is created with a set of items. Typically, it includes all the aliases that have the AllScope option. This option is discussed later in this topic. It includes all the variables that have the AllScope option, plus some variables that can be used to customize the scope, such as MaximumFunctionCount.

To find the items in a particular scope, use the Scope parameter of Get-Variable or Get-Alias.

For example, to get all the variables in the local scope, type:

```
get-variable -scope local
```

To get all the variables in the global scope, type:

```
get-variable -scope global
```

Scope Modifiers

To specify the scope of a new variable, alias, or function, use a scope modifier. The valid values of a modifier are Global, Local, Private, and Script.

The syntax for a scope modifier in a variable is:

```
$[<scope-modifier>]:<name> = <value>
```

The syntax for a scope modifier in a function is:

```
function [<scope-modifier>]:<name> {<function-body>}
```

The default scope for scripts is the script scope. The default scope for functions and aliases is the local scope, even if they are defined in a script.

The following command, which does not use a scope modifier, creates a variable in the current or local scope:

```
$a = "one"
```

To create the same variable in the global scope, use the Global scope modifier:

```
$global:a = "one"
```

To create the same variable in the script scope, use the script scope modifier:

```
$script:a = "one"
```

You can also use a scope modifier in functions. The following function definition creates a function in the global scope:

```
function global:Hello
{
write-host "Hello, World"
}
```

You can also use scope modifiers to refer to a variable in a different scope. The following command refers to the $test variable, first in the local scope and then in the global scope:

```
$test

$global:test
```

The Using scope modifier

Using is a special scope modifier that identifies a local variable in a remote command. By default, variables in remote commands are assumed to be defined in the remote session.

The Using scope modifier is introduced in Windows PowerShell 3.0.

For more information, see about_Remote_Variables.

The AllScope Option

Variables and aliases have an Option property that can take a value of AllScope. Items that have the AllScope property become part of any child scopes that you create, although they are not retroactively inherited by parent scopes.

An item that has the AllScope property is visible in the child scope, and it is part of that scope. Changes to the item in any scope affect all the scopes in which the variable is defined.

Managing Scope

Several cmdlets have a Scope parameter that lets you get or set (create and change) items in a particular scope. Use the following command to find all the cmdlets in your session that have a Scope parameter:

```
get-help * -parameter scope
```

To find the variables that are visible in a particular scope, use the Scope parameter of Get-Variable. The visible parameters include global parameters, parameters in the parent scope, and parameters in the current scope.

For example, the following command gets the variables that are visible in the local scope:

```
get-variable -scope local
```

To create a variable in a particular scope, use a scope modifier or the Scope parameter of Set-Variable. The following command creates a variable in the global scope:

```
new-variable -scope global -name a -value "One"
```

You can also use the Scope parameter of the New-Alias, Set-Alias, or Get-Alias cmdlets to specify the scope. The following command creates an alias in the global scope:

```
new-alias -scope global -name np -value Notepad.exe
```

To get the functions in a particular scope, use the Get-Item cmdlet when you are in the scope. The Get-Item cmdlet does not have a scope parameter.

Using Dot Source Notation with Scope

Scripts and functions follow all the rules of scope. You create them in a particular scope, and they affect only that scope unless you use a cmdlet parameter or a scope modifier to change that scope.

But, you can add a script or function to the current scope by using dot source notation. Then, when a script runs in the current scope, any functions, aliases, and variables that the script creates are available in the current scope.

To add a function to the current scope, type a dot (.) and a space before the path and name of the function in the function call.

For example, to run the Sample.ps1 script from the C:\Scripts directory in the script scope (the default for scripts), use the following command:

```
c:\scripts\sample.ps1
```

To run the Sample.ps1 script in the local scope, use the following command:

```
. c:\scripts.sample.ps1
```

When you use the call operator (&) to run a function or script, it is not added to the current scope. The following example uses the call operator:

```
& c:\scripts.sample.ps1
```

Any aliases, functions, or variables that the Sample.ps1 script creates are not available in the current scope.

Restricting Without Scope

A few Windows PowerShell concepts are similar to scope or interact with scope. These concepts may be confused with scope or the behavior of scope.

Sessions, modules, and nested prompts are self-contained environments, but they are not child scopes of the global scope in the session.

Sessions:

A session is an environment in which Windows PowerShell runs. When you create a session on a remote computer, Windows PowerShell establishes a persistent connection to the remote computer. The persistent connection lets you use the session for multiple related commands.

Because a session is a contained environment, it has its own scope, but a session is not a child scope of the session in which is was created. The session starts with its own global scope. This scope is independent of the global scope of the session. You can create child scopes in the session. For example, you can run a script to create a child scope in a session.

Modules:

You can use a Windows PowerShell module to share and deliver Windows PowerShell tools. A module is a unit that can contain cmdlets, scripts, functions, variables, aliases, and other useful items. Unless explicitly defined, the items in a module are not accessible outside the module. Therefore, you can add the module to your session and use the public items without worrying that the other items might override the cmdlets, scripts, functions, and other items in your session.

The privacy of a module behaves like a scope, but adding a module to a session does not change the scope. And, the module does not have its own scope, although the scripts in the module, like all Windows PowerShell scripts, do have their own scope.

Nested Prompts:

Similarly, nested prompts do not have their own scope. When you enter a nested prompt, the nested prompt is a subset of the environment. But, you remain within the local scope.

Scripts do have their own scope. If you are debugging a script, and you reach a breakpoint in the script, you enter the script scope.

Private Option:

Aliases and variables have an Option property that can take a value

of Private. Items that have the Private option can be viewed and changed in the scope in which they are created, but they cannot be viewed or changed outside that scope.

For example, if you create a variable that has a private option in the global scope and then run a script, Get-Variable commands in the script do not display the private variable. This occurs even if you use the global scope modifier.

You can use the Option parameter of the New-Variable, Set-Variable, New-Alias, and Set-Alias cmdlets to set the value of the Option property to Private.

Visibility:

The Visibility property of a variable or alias determines whether you can see the item outside the container, such as a module, in which it was created. Visibility is designed for containers in the same way that the Private value of the Option property is designed for scopes.

The Visibility property takes the Public and Private values. Items that have private visibility can be viewed and changed only in the container in which they were created. If the container is added or imported, the items that have private visibility cannot be viewed or changed.

Because Visibility is designed for containers, it works differently in a scope. If you create an item that has private visibility in the global scope, you cannot view or change the item in any scope. If you try to view or change the value of a variable that has private visibility, Windows PowerShell returns an error message.

You can use the New-Variable and Set-Variable cmdlets to create a variable that has private visibility.

EXAMPLES

Example 1: Change a Variable Value Only in a Script

The following command changes the value of the $ConfirmPreference variable in a script. The change does not affect the global scope.

First, to display the value of the $ConfirmPreference variable in the local scope, use the following command:

```
C:\PS> $ConfirmPreference
High
```

Create a Scope.ps1 script that contains the following commands:

```
$ConfirmPreference = "Low"
"The value of `$ConfirmPreference is $ConfirmPreference."
```

Run the script. The script changes the value of the $ConfirmPreference variable and then reports its value in the script scope. The output should resemble the following output:

```
The value of $ConfirmPreference is Low.
```

Next, test the current value of the $ConfirmPreference variable in the current scope.

```
C:\PS> $ConfirmPreference
High
```

This example shows that changes to the value of a variable in the script scope do not affect the value of that variable in the parent scope.

Example 2: View a Variable Value in Different Scopes

You can use scope modifiers to view the value of a variable in the local scope and in a parent scope.

First, define a $test variable in the global scope.

```
$test = "Global"
```

Next, create a Sample.ps1 script that defines the $test variable. In the script, use a scope modifier to refer to either the global or local versions of the $test variable.

```
# In Sample.ps1
```

```
$test = "Local"
"The local value of `$test is $test."
"The global value of `$test is $global:test."
```

When you run Sample.ps1, the output should resemble the following output:

```
The local value of $test is Local.
The global value of $test is Global.
```

When the script is complete, only the global value of $test is defined in the session.

```
C:\PS> $test
Global
```

Example 3: Change the Value of a Variable in a Parent Scope

Unless you protect an item by using the Private option or another method, you can view and change the value of a variable in a parent scope.

First, define a $test variable in the global scope.

```
$test = "Global"
```

Next, create a Sample.ps1 script that defines the $test variable. In the script, use a scope modifier to refer to either the global or local versions of the $test variable.

```
# In Sample.ps1

$global:test = "Local"
"The global value of `$test is $global:test."
```

When the script is complete, the global value of $test is changed.

```
C:\PS> $test
Local
```

Example 4: Creating a Private Variable

A private variable is a variable that has an Option property that has a value of Private. Private variables are inherited by the child scope, but they can be viewed or changed only in the scope in which they were created.

The following command creates a private variable called $ptest in the local scope.

```
New-Variable -Name ptest -Value 1 -Option private
```

You can display and change the value of $ptest in the local scope.

```
C:\PS> $ptest
      1
      C:\PS> $ptest = 2
      C:\PS> $ptest
2
```

Next, create a Sample.ps1 script that contains the following commands. The command tries to display and change the value of $ptest.

```
    # In Sample.ps1

    "The value of `$Ptest is $Ptest."
  "The value of `$Ptest is $global:Ptest."
```

Because the $ptest variable is not visible in the script scope, the output is empty.

```
    "The value of $Ptest is ."
  "The value of $Ptest is ."
```

Example 5: Using a Local Variable in a Remote Command

To indicate that a variable in a remote command was created in the local session, use the Using scope modifier. By default, Windows PowerShell assumes that the variables in remote commands were created in the remote session.

The syntax is:

```
    $Using:<VariableName>
```

For example, the following commands create a $Cred variable in the local session and then use the $Cred variable in a remote command:

```
$Cred = Get-Credential
Invoke-Command $s {Remove-Item .\Test*.ps1 -Credential $Using:Cred}
```

The Using scope was introduced in Windows PowerShell 3.0.

In Windows PowerShell 2.0, to indicate that a variable was created in the local session, use the following command format.

```
$Cred = Get-Credential
Invoke-Command $s {param($c) Remove-Item .\Test*.ps1 -Credential $c} -ArgumentList $Cred
```

SEE ALSO

about_Variables
about_Environment_Variables
about_Functions
about_Script_Blocks

TOPIC

about_Scripts

SHORT DESCRIPTION

Describes how to run and write scripts in Windows PowerShell.

LONG DESCRIPTION

A script is a plain text file that contains one or more Windows PowerShell commands. Windows PowerShell scripts have a .ps1 file name extension.

Running a script is a lot like running cmdlet. You type the path and file name of the script and use parameters to submit data and set options. You can run scripts on your computer or in a remote session on a different

computer.

Writing a script saves a command for later use and makes it easy to share with others. Most importantly, it lets you run the commands simply by typing the script path and the file name. Scripts can be as simple as a single command in a file or as extensive as a complex program.

Scripts have additional features, such as the #Requires special comment, the use of parameters, support for data sections, and digital signing for security. You can also write Help topics for scripts and for any functions in the script.

HOW TO RUN A SCRIPT

Before you can run a script, you need to change the default Windows PowerShell execution policy. The default execution policy, "Restricted", prevents all scripts from running, including scripts that you write on the local computer. For more information, see about_Execution_Policies.

The execution policy is saved in the registry, so you need to change it only once on each computer.

To change the execution policy, use the following procedure.

1. Start Windows PowerShell with the "Run as administrator" option.

2. At the command prompt, type:

   Set-ExecutionPolicy AllSigned
-or-
   Set-ExecutionPolicy RemoteSigned

The change is effective immediately

To run a script, type the full name and the full path to the script file.

For example, to run the Get-ServiceLog.ps1 script in the C:\Scripts directory, type:

C:\Scripts\Get-ServiceLog.ps1

To run a script in the current directory, type the path to the current directory, or use a dot to represent the current directory, followed by a path backslash (.\).

For example, to run the ServicesLog.ps1 script in the local directory, type:

```
.\Get-ServiceLog.ps1
```

If the script has parameters, type the parameters and parameter values after the script file name.

For example, the following command uses the ServiceName parameter of the Get-ServiceLog script to request a log of WinRM service activity.

```
.\Get-ServiceLog.ps1 -ServiceName WinRM
```

As a security feature, Windows PowerShell does not run scripts when you double-click the script icon in File Explorer or when you type the script name without a full path, even when the script is in the current directory. For more information about running commands and scripts in Windows PowerShell, see about_Command_Precedence.

RUN WITH POWERSHELL

Beginning in Windows PowerShell 3.0, you can run scripts from File Explorer (or Windows Explorer, in earlier versions of Windows).

To use the "Run with PowerShell" feature:

In File Explorer (or Windows Explorer), right-click the script file name and then select "Run with PowerShell".

The "Run with PowerShell" feature is designed to run scripts that do not have required parameters and do not return output to the command prompt.

For more information, see about_Run_With_PowerShell

RUNNING SCRIPTS ON OTHER COMPUTERS

To run a script on one or more remote computers, use the FilePath parameter of the Invoke-Command cmdlet.

Enter the path and file name of the script as the value of the FilePath parameter. The script must reside on the local computer or in a directory that the local computer can access.

The following command runs the Get-ServiceLog.ps1 script on the Server01 and Server02 remote computers.

```
Invoke-Command -ComputerName Server01, Server02 -FilePath C:\Scripts\Get-ServiceLog.ps1
```

GET HELP FOR SCRIPTS

The Get-Help cmdlet gets the help topics for scripts as well as for cmdlets and other types of commands. To get the help topic for a script, type "Get-Help" followed by the path and file name of the script. If the script path is in your Path environment variable, you can omit the path.

For example, to get help for the ServicesLog.ps1 script, type:

```
get-help C:\admin\scripts\ServicesLog.ps1
```

HOW TO WRITE A SCRIPT

A script can contain any valid Windows PowerShell commands, including single commands, commands that use the pipeline, functions, and control structures such as If statements and For loops.

To write a script, start a text editor (such as Notepad) or a script editor (such as the Windows PowerShell Integrated Scripting Environment [ISE]). Type the commands and save them in a file with a valid file name and the .ps1 file name extension.

The following example is a simple script that gets the services that are running on the current system and saves them to a log file. The log file name is created from the current date.

```
$date = (get-date).dayofyear
get-service | out-file "$date.log"
```

To create this script, open a text editor or a script editor, type these commands, and then save them in a file named ServiceLog.ps1.

PARAMETERS IN SCRIPTS

To define parameters in a script, use a Param statement. The Param statement must be the first statement in a script, except for comments and any #Requires statements.

Script parameters work like function parameters. The parameter values are available to all of the commands in the script. All of the features of

function parameters, including the Parameter attribute and its named arguments, are also valid in scripts.

When running the script, script users type the parameters after the script name.

The following example shows a Test-Remote.ps1 script that has a ComputerName parameter. Both of the script functions can access the ComputerName parameter value.

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

function CanPing {
  $error.clear()
  $tmp = test-connection $computername -erroraction SilentlyContinue

  if (!$?)
     {write-host "Ping failed: $ComputerName."; return $false}
  else
     {write-host "Ping succeeded: $ComputerName"; return $true}
}

function CanRemote {
   $s = new-pssession $computername -erroraction SilentlyContinue

   if ($s -is [System.Management.Automation.Runspaces.PSSession])
      {write-host "Remote test succeeded: $ComputerName."}
   else
      {write-host "Remote test failed: $ComputerName."}
}

if (CanPing $computername) {CanRemote $computername}
```

To run this script, type the parameter name after the script name. For example:

```
C:\PS> .\test-remote.ps1 -computername Server01

Ping succeeded: Server01
Remote test failed: Server01
```

For more information about the Param statement and the function parameters, see about_Functions and about_Functions_Advanced_Parameters.

WRITING HELP FOR SCRIPTS

You can write a help topic for a script by using either of the two following methods:

-- Comment-Based Help for Scripts

Create a Help topic by using special keywords in the comments. To create comment-based Help for a script, the comments must be placed at the beginning or end of the script file. For more information about comment-based Help, see about_Comment_Based_Help.

-- XML-Based Help for Scripts

Create an XML-based Help topic, such as the type that is typically created for cmdlets. XML-based Help is required if you are translating Help topics into multiple languages.

To associate the script with the XML-based Help topic, use the .ExternalHelp Help comment keyword. For more information about the ExternalHelp keyword, see about_Comment_Based_Help. For more information about XML-based help, see "How to Write Cmdlet Help" in the MSDN (Microsoft Developer Network) library at http://go.microsoft.com/fwlink/?LinkID=123415.

SCRIPT SCOPE AND DOT SOURCING

Each script runs in its own scope. The functions, variables, aliases, and drives that are created in the script exist only in the script scope. You cannot access these items or their values in the scope in which the script runs.

To run a script in a different scope, you can specify a scope, such as Global or Local, or you can dot source the script.

The dot sourcing feature lets you run a script in the current scope instead of in the script scope. When you run a script that is dot sourced, the commands in the script run as though you had typed them at the command prompt. The functions, variables, aliases, and drives that the script creates are created in the scope in which you are working. After the script runs, you can use the created items and access their values in your session.

To dot source a script, type a dot (.) and a space before the script path.

For example:

```
. C:\scripts\UtilityFunctions.ps1
```

-or-

```
. .\UtilityFunctions.ps1
```

After the UtilityFunctions script runs, the functions and variables that the script creates are added to the current scope.

For example, the UtilityFunctions.ps1 script creates the New-Profile function and the $ProfileName variable.

```
#In UtilityFunctions.ps1

function New-Profile
{
   Write-Host "Running New-Profile function"
   $profileName = split-path $profile -leaf

   if (test-path $profile)
     {write-error "There is already a $profileName profile on this computer."}
   else
{new-item -type file -path $profile -force }
}
```

If you run the UtilityFunctions.ps1 script in its own script scope, the New-Profile function and the $ProfileName variable exist only while the script is running. When the script exits, the function and variable are removed, as shown in the following example.

```
C:\PS> .\UtilityFunctions.ps1

C:\PS> New-Profile
The term 'new-profile' is not recognized as a cmdlet, function, operable
program, or script file. Verify the term and try again.
At line:1 char:12
+ new-profile <<<<
  + CategoryInfo          : ObjectNotFound: (new-profile:String) [],
  + FullyQualifiedErrorId : CommandNotFoundException

C:\PS> $profileName
C:\PS>
```

When you dot source the script and run it, the script creates the New-Profile function and the $ProfileName variable in your session in your scope. After the script runs, you can use the New-Profile function in your session, as shown in the following example.

```
C:\PS> . .\UtilityFunctions.ps1

C:\PS> New-Profile


    Directory: C:\Users\juneb\Documents\WindowsPowerShell


Mode    LastWriteTime     Length Name
----    -------------     ------ ----
-a---   1/14/2009 3:08 PM      0 Microsoft.PowerShellISE_profile.ps1


C:\PS> $profileName
Microsoft.PowerShellISE_profile.ps1
```

For more information about scope, see about_Scopes.

SCRIPTS IN MODULES

A module is a set of related Windows PowerShell resources that can be distributed as a unit. You can use modules to organize your scripts, functions, and other resources. You can also use modules to distribute your code to others, and to get code from trusted sources.

You can include scripts in your modules, or you can create a script module, which is a module that consists entirely or primarily of a script and supporting resources. A script module is just a script with a .psm1 file name extension.

For more information about modules, see about_Modules.

OTHER SCRIPT FEATURES

Windows PowerShell has many useful features that you can use in scripts.

#Requires
You can use a #Requires statement to prevent a script from running without specified modules or snap-ins and a specified version of Windows PowerShell. For more information, see about_Requires.

$PSCommandPath
Contains the full path and name of the script that is being run. This parameter is valid in all scripts. This automatic variable is

introduced in Windows PowerShell 3.0.

$PSScriptRoot
Contains the directory from which a script is being run. In Windows PowerShell 2.0, this variable is valid only in script modules (.psm1). Beginning in Windows PowerShell 3.0, it is valid in all scripts.

$MyInvocation
The $MyInvocation automatic variable contains information about the current script, including information about how it was started or "invoked." You can use this variable and its properties to get information about the script while it is running. For example, the $MyInvocation.MyCommand.Path variable contains the path and file name of the script. $MyInvocation.Line contains the command that started the script, including all parameters and values.

Beginning in Windows PowerShell 3.0, $MyInvocation has two new properties that provide information about the script that called or invoked the current script. The values of these properties are populated only when the invoker or caller is a script.

-- PSCommandPath contains the full path and name of the script that called or invoked the current script.
-- PSScriptRoot contains the directory of the script that called or invoked the current script.

Unlike the $PSCommandPath and $PSScriptRoot automatic variables, which contain information about the current script, the PSCommandPath and PSScriptRoot properties of the $MyInvocation variable contain information about the script that called or invoke the current script.

Data sections
You can use the Data keyword to separate data from logic in scripts. Data sections can also make localization easier. For more information, see about_Data_Sections and about_Script_Localization.

Script Signing
You can add a digital signature to a script. Depending on the execution policy, you can use digital signatures to restrict the running of scripts that could include unsafe commands. For more information, see about_Execution_Policies and about_Signing.

SEE ALSO

about_Command_Precedence
about_Comment_Based_Help
about_Execution_Policies
about_Functions
about_Modules
about_Profiles
about_Requires
about_Run_With_PowerShell
about_Scopes
about_Script_Blocks
about_Signing
Invoke-Command

TOPIC

about_Script_Blocks

SHORT DESCRIPTION

Defines what a script block is and explains how to use script blocks in the Windows PowerShell programming language.

LONG DESCRIPTION

In the Windows PowerShell programming language, a script block is a collection of statements or expressions that can be used as a single unit. A script block can accept arguments and return values.

Syntactically, a script block is a statement list in braces, as shown in the following syntax:

```
{<statement list>}
```

A script block returns the output of all the commands in the script block, either as a single object or as an array.

Like functions, a script block can include parameters. Use the Param keyword to assign named parameters, as shown in the following syntax:

```
{
   param ([type]$parameter1 [,[type]$parameter2])
   <statement list>
}
```

In a script block, unlike a function, you cannot specify parameters outside the braces.

Like functions, script blocks can include the DynamicParam, Begin, Process, and End keywords. For more information, see about_Functions and about_Functions_Advanced.

Using Script Blocks

A script block is an instance of a Microsoft .NET Framework type (System.Management.Automation.ScriptBlock). Commands can have script block parameter values. For example, the Invoke-Command cmdlet has a ScriptBlock parameter that takes a script block value, as shown in this example:

```
C:\PS> invoke-command -scriptblock  { get-process }
Handles  NPM(K)    PM(K)     WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----     ----- -----   ------     -- -----------
   999     28    39100    45020   262    15.88   1844 communicator
   721     28    32696    36536   222    20.84   4028 explorer
. . .
```

The script block that is used as a value can be more complicated, as shown in the following example:

```
C:\PS> invoke-command -scriptblock { param ($uu = "Parameter");
   "$uu assigned." }
Parameter assigned.
```

The script block in the preceding example uses the Param keyword to create a parameter that has a default value. The following example uses

the Args parameter of the Invoke-Command cmdlet to assign a different value to the parameter:

```
C:\PS> invoke-command -scriptblock {param ($uu = "Parameter");
   "$uu assigned."} -args "Other value"
Other value assigned.
```

You can assign a script block to a variable, as shown in the following example:

```
C:\PS> $a = {param ($uu = "Parameter"); "$uu assigned."}
```

You can use the variable with a cmdlet such as Invoke-Command, as shown in the following example:

```
C:\PS> invoke-command -scriptblock $a -args "Other value"
Other value assigned.
```

You can run a script block that is assigned to a variable by using the call operator (&), as shown in the following example:

```
C:\PS> &$a
Parameter assigned.
```

You can also provide a parameter to the script block, as shown in the following example:

```
C:\PS> &$a "Other value"
Other value assigned.
```

If you want to assign the value that is created by a script block to a variable, use the call operator to run the script block directly, as shown in the following example:

```
C:\PS> $a = &{param ($uu = "Parameter"); "$uu assigned."}
C:\PS> $a
Parameter assigned.
```

For more information about the call operator, see about_Operators.

SEE ALSO

about_Functions
about_Functions_Advanced
about_Operators

TOPIC

about_Script_Internationalization

SHORT DESCRIPTION

Describes the script internationalization features of Windows PowerShell 2.0 that make it easy for scripts to display messages and instructions to users in their user interface (UI) language.

LONG DESCRIPTION

The Windows PowerShell script internationalization features allow you to better serve users throughout the world by displaying Help and user messages for scripts and functions in the user's UI language.

The script internationalization features query the UI culture of the operating system during execution, import the appropriate translated text strings, and display them to the user. The Data section lets you store text strings separate from code so they are easily identified and extracted. A new cmdlet, ConvertFrom-StringData, converts text strings into dictionary-like hash tables to facilitate translation.

The Windows PowerShell 2.0 features used in script internationalization

are not supported by Windows PowerShell 1.0. Scripts that include these features will not run in Windows PowerShell 1.0 without modification.

To support international Help text, Windows PowerShell 2.0 includes the following features:

-- A Data section that separates text strings from code instructions. For more information about the Data section, see about_Data_Sections.

-- New automatic variables, $PSCulture and $PSUICulture. $PSCulture stores the name of the UI language used on the system for elements such as the date, time, and currency. The $PSUICulture variable stores the name of the UI language used on the system for user interface elements such as menus and text strings.

-- A cmdlet, ConvertFrom-StringData, that converts text strings into dictionary-like hash tables to facilitate translation. For more information, see ConvertFrom-StringData.

-- A new file type, .psd1, that stores translated text strings. The .psd1 files are stored in language-specific subdirectories of the script directory.

-- A cmdlet, Import-LocalizedData, that imports translated text strings for a specified language into a script at runtime. This cmdlet recognizes and imports strings in any Windows-supported language. For more information see Import-LocalizedData.

THE DATA SECTION: Storing Default Strings

Use a Data section in the script to store the text strings in the default language. Arrange the strings in key/value pairs in a here-string. Each key/value pair must be on a separate line. If you include comments, the comments must be on separate lines.

The ConvertFrom-StringData cmdlet converts the key/value pairs in the here-string into a dictionary-like hash table that is stored in the value of the Data section variable.

In the following example, the Data section of the World.ps1 script includes the English-United States (en-US) set of prompt messages for a script. The ConvertFrom-StringData cmdlet converts the strings into a hash table and stores them in the $msgtable variable.

```
$msgTable = Data {
  # culture="en-US"
  ConvertFrom-StringData @'
```

```
helloWorld = Hello, World.
      errorMsg1 = You cannot leave the user name field blank.
       promptMsg = Please enter your user name.
'@
}
```

For more information about here-strings, see about_Quoting_Rules.

PSD1 FILES: Storing Translated Strings

Save the script messages for each UI language in separate text files with the same name as the script and the .psd1 file name extension. Store the files in subdirectories of the script directory with names of cultures in the following format:

<language>–<region>

Examples: de-DE, ar-SA, and zh-Hans

For example, if the World.ps1 script is stored in the C:\Scripts directory, you would create a file directory structure that resembles the following:

```
C:\Scripts
C:\Scripts\World.ps1
C:\Scripts\de-DE\World.psd1
      C:\Scripts\ar-SA\World.psd1
      C:\Scripts\zh-CN\World.psd1
...
```

The World.psd1 file in the de-DE subdirectory of the script directory might include the following statement:

```
ConvertFrom-StringData @'
helloWorld = Hello, World (in German).
errorMsg1 = You cannot leave the user name field blank (in German).
      promptMsg = Please enter your user name (in German).
'@
```

Similarly, the World.psd1 file in the ar-SA subdirectory of the script directory might includes the following statement:

```
   ConvertFrom-StringData @'
   helloWorld = Hello, World (in Arabic).
   errorMsg1 = You cannot leave the user name field blank (in Arabic).
         promptMsg = Please enter your user name (in Arabic).
'@
```

IMPORT-LOCALIZEDDATA: Dynamic Retrieval of Translated Strings

To retrieve the strings in the UI language of the current user, use the Import-LocalizedData cmdlet.

Import-LocalizedData finds the value of the $PSUICulture automatic variable and imports the content of the <script-name>.psd1 files in the subdirectory that matches the $PSUICulture value. Then, it saves the imported content in the variable specified by the value of the BindingVariable parameter.

```
import-localizeddata -bindingVariable msgTable
```

For example, if the Import-LocalizedData command appears in the C:\Scripts\World.ps1 script and the value of $PSUICulture is "ar-SA", Import-LocalizedData finds the following file:

```
    C:\Scripts\ar-SA\World.psd1
```

Then, it imports the Arabic text strings from the file into the $msgTable variable, replacing any default strings that might be defined in the Data section of the World.ps1 script.

As a result, when the script uses the $msgTable variable to display user messages, the messages are displayed in Arabic.

For example, the following script displays the "Please enter your user name" message in Arabic:

```
if (!($username)) { $msgTable.promptMsg }
```

If Import-LocalizedData cannot find a .psd1 file that matches the value of $PSUIculture, the value of $msgTable is not replaced, and the call to $msgTable.promptMsg displays the fallback en-US strings.

ExAMPLE

This example shows how the script internationalization features
are used in a script to display a day of the week to users
in the language that is set on the computer.

The following is a complete listing of the Sample1.ps1 script
file.

The script begins with a Data section named Day ($Day) that
contains a ConvertFrom-StringData command. The expression
submitted to ConvertFrom-StringData is a here-string that
contains the day names in the default UI culture, en-US, in
key/value pairs. The ConvertFrom-StringData cmdlet converts
the key/value pairs in the here-string into a hash table and
then saves it in the value of the $Day variable.

The Import-LocalizedData command imports the contents of the
.psd1 file in the directory that matches the value of the
$PSUICulture automatic variable and then saves it in the $Day
variable, replacing the values of $Day that are defined in the
Data section.

The remaining commands load the strings into an array and display
them.

```
    $Day = DATA {
# culture="en-US"
ConvertFrom-StringData @'
    messageDate = Today is
  d0 = Sunday
  d1 = Monday
  d2 = Tuesday
  d3 = Wednesday
  d4 = Thursday
  d5 = Friday
  d6 = Saturday
    '@
}


Import-LocalizedData -BindingVariable Day

# Build an array of weekdays.
$a = $Day.d0, $Day.d1, $Day.d2, $Day.d3, $Day.d4, $Day.d5, $Day.d6


    # Get the day of the week as a number (Monday = 1).
    # Index into $a to get the name of the day.
    # Use string formatting to build a sentence.
```

```
    "{0} {1}" –f $Day.messageDate, $a[(get-date -uformat %u)] | Out-Host
```

The .psd1 files that support the script are saved in subdirectories of the script directory with names that match the $PSUICulture values.

The following is a complete listing of .\de-DE\sample1.psd1:

```
    # culture="de-DE"
ConvertFrom-StringData @'
  messageDate = Today is
      d0 = Sunday (in German)
  d1 = Monday (in German)
  d2 = Tuesday (in German)
  d3 = Wednesday (in German)
  d4 = Thursday (in German)
      d5 = Friday (in German)
      d6 = Saturday (in German)
    '@
```

As a result, when you run Sample.ps1 on a system on which the value of $PSUICulture is de-DE, the output of the script is:

```
Today is Friday (in German)
```

SEE ALSO

- about_Data_Sections
- about_Automatic_Variables
- about_Hash_Tables
- about_Quoting_Rules
- ConvertFrom-StringData
- Import-LocalizedData

```
Name                Category  Module            Synopsis
----                --------  ------            --------
about_Sequence      HelpFile                    Describes the Sequence keyword, which runs
selected
about_Sequence      HelpFile                    Describes the Sequence keyword, which runs
selected
```

TOPIC

about_Session_Configurations

SHORT DESCRIPTION

Describes session configurations, which determine the users who can connect to the computer remotely and the commands they can run.

LONG DESCRIPTION

A session configuration, also known as an "endpoint" is a group of settings on the local computer that define the environment for the Windows PowerShell sessions that are created when remote or local users connect to Windows PowerShell on the local computer.

Administrators of the computer can use session configurations to protect the computer and to define custom environments for users who connect to the computer.

Administrators can also use session configurations to determine the permissions that are required to connect to the computer remotely. By default, only members of the Administrators group have permission to use the session configuration to connect remotely, but you can change the default settings to allow all users, or selected users, to connect remotely to your computer.

Beginning in Windows PowerShell 3.0, you can use a session configuration file to define the elements of a session configuration. This feature makes it easy to customize sessions without writing code and to discover the properties of a session configuration. To create a session configuration file, use the New-PSSessionConfiguration cmdlet. For more information about session configuration files, see about_Session_Configuration_Files (http://go.microsoft.com/fwlink/?LinkId=236023.

Session configurations are a feature of Web Services for Management (WS-Management) based Windows PowerShell remoting. They are used only when you use the New-PSSession, Invoke-Command, or Enter-PSSession cmdlets to connect to a remote computer.

Note: To manage the session configurations, start Windows PowerShell with the "Run as administrator" option.

About Session Configurations

Every Windows PowerShell session uses a session configuration. This includes persistent sessions that you create by using the New-PSSession or Enter-PSSession cmdlets, and the temporary sessions that Windows PowerShell creates when you use the ComputerName parameter of a cmdlet that uses WS-Management-based remoting technology, such as Invoke-Command.

Administrators can use session configurations to protect the resources of the computer and to create custom environments for users who connect to the computer. For example, you can use a session configuration to limit the size of objects that the computer receives in the session, to define the language mode of the session, and to specify the cmdlets, providers, and functions that are available in the session.

By configuring the security descriptor of a session configuration, you determine who can use the session configuration to connect to the computer. Users must have Execute permission to a session configuration to use it in a session. If a user does not have the required permissions to use any of the session configurations on a computer, the user cannot connect to the computer remotely.

By default, only Administrators of the computer have permission to use the default session configurations. But, you can change the security descriptors to allow everyone, no one, or only selected users to use the session configurations on your computer.

Built-in Session Configurations

Windows PowerShell 3.0 includes built-in session configurations named Microsoft.PowerShell and Microsoft.PowerShell.Workflow. On computers running 64-bit versions of Windows, Windows PowerShell also provides Microsoft.PowerShell32, a 32-bit session configuration.

The Microsoft.PowerShell session configuration is used for sessions by default, that is, when a command to create a session does not include the ConfigurationName parameter of the New-PSSession, Enter-PSSession, or Invoke-Command cmdlet.

The security descriptors for the default session configurations allow

only members of the Administrators group on the local computer to use them. As such, only members of the Administrators group can connect to the computer remotely unless you change the default settings.

You can change the default session configurations by using the $PSSessionConfigurationName preference variable. For more information, see about_Preference_Variables.

Viewing Session Configurations on the Local Computer

To get the session configurations on your local computer, use the Get-PSSessionConfiguration cmdlet.

For example, type:

```
PS C:\> Get-PSSessionConfiguration | Format-List -Property Name, Permission

Name       : microsoft.powershell
Permission : BUILTIN\Administrators AccessAllowed

Name       : microsoft.powershell.workflow
Permission : BUILTIN\Administrators AccessAllowed

Name       : microsoft.powershell32
Permission : BUILTIN\Administrators AccessAllowed
```

The session configuration object is expanded in Windows PowerShell 3.0 to display the properties of the session configuration that are configured by using a session configuration file.

For example, to see all of the properties of a session configuration object, type:

```
PS C:\> Get-PSSessionConfiguration | Format-List -Property *
```

You can also use the WSMan provider in Windows PowerShell to view session configurations. The WSMan provider creates a WSMAN: drive in your session.

In the WSMAN: drive, session configurations are in the Plugin node. (All session configurations are in the Plugin node, but there are items in the Plugin node that are not session configurations.)

For example, to view the session configurations on the local computer, type:

```
PS C:\> dir wsman:\localhost\plugin\microsoft*

   WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin

Type          Keys                        Name
----          ----                        ----
Container     {Name=microsoft.powershell}       microsoft.powershell
Container     {Name=microsoft.powershell.workf... microsoft.powershell.workflow
Container     {Name=microsoft.powershell32}     microsoft.powershell32
```

Viewing Session Configurations on a Remote Computer

To view the session configurations on a remote computer, use the Connect-WSMan cmdlet to add a note for the remote computer to the WSMAN: drive on your local computer, and then use the WSMAN: drive to view the session configurations.

For example, the following command adds a node for the Server01 remote computer to the WSMAN: drive on the local computer.

```
PS C:\> Connect-WSMan server01.corp.fabrikam.com
```

When the command is complete, you can navigate to the node for the Server01 computer to view the session configurations.

For example:

```
PS C:\> cd wsman:

PS WSMan:\> dir

ComputerName                    Type
------------                    ----
localhost                       Container
server01.corp.fabrikam.com      Container

PS WSMan:\> dir server01*\plugin\*


    WSManConfig: Microsoft.WSMan.Management\WSMan::server01.corp.fabrikam.com\Plugin


Type          Keys                        Name
----          ----                        ----
Container     {Name=microsoft.powershell}       microsoft.powershell
Container     {Name=microsoft.powershell.workf... microsoft.powershell.workflow
Container     {Name=microsoft.powershell32}     microsoft.powershell32
```

Changing the Security Descriptor of a Session Configuration

In Windows Server 2012 and newer releases of Windows Server, the built-in session configurations are enabled for remote users by default. In other supported versions of Windows, you must change the security descriptors of the session configurations to allow remote access.

To enable remote access to the session configurations on the computer, use the Enable-PSRemoting cmdlet.

Also, by default, only members of the Administrators group on the computer have Execute permission to the default session configurations, but you can change the security descriptors on the default session configurations and on any session configurations that you create.

To give other users permission to connect to the computer remotely, use the Set-PSSessionConfiguration cmdlet to add "Execute" permissions for those users to the security descriptors of the Microsoft.PowerShell and Microsoft.PowerShell32 session configurations.

For example, the following command opens a property page that lets you change the security descriptor for the Microsoft.PowerShell default session configuration.

```
PS C:\> Set-PSSessionConfiguration -name Microsoft.PowerShell -ShowSecurityDescriptorUI
```

To deny everyone permission to all the session configurations on the computer, use the Disable-PSSessionConfiguration cmdlet. For example, the following command disables the default session configurations on the computer.

```
PS C:\> Disable-PSSessionConfiguration -Name Microsoft.PowerShell
```

To prevent remote users from connecting to the computer, but allow local users to connect, use the Disable-PSRemoting cmdlet. Disable-PSRemoting adds a "Network_Deny_All" entry to all session configurations on the computer.

```
PS C:\> Disable-PSRemoting
```

To allow remote users to use all session configurations on the computer, use the Enable-PSRemoting or Enable-PSSessionConfiguration cmdlet. For example, the following command enables remote access to the built-in session configurations.

```
PS C:\> Enable-PSSessionConfiguration -name Microsoft.Power*
```

To make other changes to the security descriptor of a session configuration, use the Set-PSSessionConfiguration cmdlet. Use the SecurityDescriptorSDDL parameter to submit an SDDL string value. Use the ShowSecurityDescriptorUI parameter to display a user interface property sheet that helps you to create a new SDDL.

For example:

```
PS C:\> Set-PSSessionConfiguration -Name Microsoft.PowerShell -ShowSecurityDescriptorUI
```

Creating a New Session Configuration

To create a new session configuration on the local computer, use the Register-PSSessionConfiguration cmdlet. To define the new session configuration, you can use a C# assembly, a Windows PowerShell script, and the parameters of the Register-PSSessionConfiguration cmdlet.

For example, the following command creates a session configuration that is identical the Microsoft.PowerShell session configuration, except that it limits the data received from a remote command to 20 megabytes (MB). (The default is 50 MB).

```
PS C:\> Register-PSSessionConfiguration -Name NewConfig --
MaximumReceivedDataSizePerCommandMB                20
```

When you create a session configuration, you can manage it by using the other session configuration cmdlets, and it appears in the WSMAN: drive.

For more information, see Register-PSSessionConfiguration.

Removing a Session Configuration

To remove a session configuration from the local computer, use the Unregister-PSSessionConfiguration cmdlet. For example, the following command removes the NewConfig session configuration from the computer.

```
PS C:\> Unregister-PSSessionConfiguration -Name NewConfig
```

For more information, see Unregister-PSSessionConfiguration.

Restoring a Session Configuration

To restore a default session configuration that was deleted (unregistered) accidentally, use the Enable-PSRemoting cmdlet.

The Enable-PSRemoting cmdlet recreates all default sessions configurations that do not exist on the computer. It does not overwrite or change the property values of existing session configurations.

To restore the original property values of a default session configuration, use the Unregister-PSSessionConfiguration to delete the session configuration and then use the Enable-PSRemoting cmdlet to recreate it.

Selecting a Session Configuration

To select a particular session configuration for a session, use the ConfigurationName parameter of New-PSSession, Enter-PSSession, or Invoke-Command.

For example, this command uses the New-PSSession cmdlet to start a PSSession on the Server01 computer. The command uses the ConfigurationName parameter to select the WithProfile configuration on the Server01 computer.

```
PS C:\> New-PSSession -ComputerName Server01 -ConfigurationName WithProfile
```

This command will succeed only if the current user has permission to use the WithProfile session configuration or can supply the credentials of a user who has the required permissions.

You can also use the $PSSessionConfigurationName preference variable to change the default session configuration on the computer. For more information about the $PSSessionConfigurationName preference variable, see about_Preference_Variables.

KEYWORDS
- about_Endpoints
- about_SessionConfigurations

SEE ALSO
- about_Preference_Variables
- about_PSSession
- about_Remote
- about_Session_Configuration_Files
- New-PSSession
- Disable-PSSessionConfiguration
- Enable-PSSessionConfiguration

Get-PSSessionConfiguration
New-PSSessionConfigurationFile
Register-PSSessionConfiguration
Set-PSSessionConfiguration
Test-PSSessionConfigurationFile
Unregister-PSSessionConfiguration

TOPIC

about_Session_Configuration_Files

SHORT DESCRIPTION

Describes session configuration files, which can be used in a session configuration ("endpoint") to define the environment of sessions that use the session configuration.

LONG DESCRIPTION

A "session configuration file" is a text file with a .pssc file name extension that contains a hash table of session configuration properties and values. You can use a session configuration file to set the properties of a session configuration and, thereby, to define the environment of Windows PowerShell sessions that use the session configuration.

Session configuration files make it easy to design custom session configurations without complex C# assemblies or scripts.

A "session configuration" or "endpoint" is a collection of settings on the local computer that determine which users can create sessions on the computer and which commands they can run in the sessions. For more information about session configurations, see about_Session_Configurations (http://go.microsoft.com/fwlink/?LinkID=145152).

Session configurations were introduced in Windows PowerShell 2.0. Session configuration files were introduced in Windows PowerShell 3.0. You must use Windows PowerShell 3.0 to include a session configuration file in a session configuration, but users of Windows PowerShell 2.0 and later are affected by all settings in the session configuration.

Creating Custom Sessions

You can customize many features of a Windows PowerShell session by specifying session properties in a session configuration. You can customize a session by writing a C# program that defines a custom runspace, or you can use a session configuration file to define the properties of sessions that are created by using the session configuration.

You can use a session configuration file to create fully functioning sessions for highly trusted users, locked-down sessions for minimal access, and sessions designed for particular tasks that contain only the modules required for the task.

For example, you can determine whether users of the session can use Windows PowerShell language elements, such as script blocks, or whether they can only run commands. You can determine which version of Windows PowerShell can run in the session, which modules are imported into the session and which cmdlets, functions, and aliases session users can run.

Creating a Session Configuration File

The easiest way to create a session configuration file is by using the New-PSSessionConfiguration cmdlet. This cmdlet generates a file with the correct syntax and format, and it verifies many of the property values.

For detailed descriptions of the properties that you can set in a session configuration file, see the help topic for the New-PSSessionConfigurationFile cmdlet.

To create a session configuration file with the default values, use the following command:

```
PS C:\> New-PSSessionConfigurationFile -Path .\Defaults.pssc
```

To open and view the file in your default text editor, use the following command:

```
PS C:\> Invoke-Item -Path .\Defaults.pssc
```

To create a session configuration for sessions in which user can run commands, but not use other elements of the Windows PowerShell language, type:

```
PS C:\> New-PSSessionConfigurationFile -LanguageMode NoLanguage -Path .\NoLanguage.pssc
```

To create a session configuration for sessions in which users can use only Get cmdlets, type:

```
PS C:\> New-PSSessionConfigurationFile -VisibleCmdlets Get-* -Path .\GetSessions.pssc
```

Using a Session Configuration File

You can include a session configuration file when you create a session configuration or add it to the session configuration at later time.

To include a session configuration file when creating a session configuration, use the Path parameter of the Register-PSSessionConfiguration cmdlet.

The following command includes the NoLanguage.pssc file when it creates the NoLanguage session configuration.

```
PS C:\> Register-PSSessionConfiguration -Name NoLanguage -Path .\NoLanguage.pssc
```

To add a session configuration file to an existing session configuration, use the Path parameter of the Set-PSSessionConfiguration cmdlet. The change affects all new sessions created with the session configuration after the command completes.

The following command adds the NoLanguage.pssc file to LockedDown session configuration.

```
PS C:\> Set-PSSessionConfiguration -Name LockedDown -Path .\NoLanguage.pssc
```

When users use the LockedDown session configuration to create a session, they can run cmdlets, but they cannot create or use variables, assign values, or use other Windows PowerShell language elements.

For example, the following command uses the New-PSSession cmdlet to create a session on the local computer that uses the LockedDown session configuration. The command saves the session in the $s variable. The ACL of the session configuration determines who can use it to create a session.

```
PS C:\> $s = New-PSSession -ComputerName Srv01 -ConfigurationName LockedDown
```

The following command uses the Invoke-Command cmdlet to run commands in the session in the $s variable. The first command, which runs the Get-UICulture cmdlet, succeeds. However, the second command, which gets the value of the $PSUICulture variable, fails.

```
PS C:\> Invoke-Command -Session $s {Get-UICulture}
en-US

PS C:\> Invoke-Command -Session $s {$PSUICulture}
The syntax is not supported by this runspace. This might be because it is in no-language mode.
  + CategoryInfo          : ParserError: ($PSUICulture:String) [], ParseException
  + FullyQualifiedErrorId : ScriptsNotAllowed
```

Editing a Session Configuration File

To edit the session configuration file that is being used by a session configuration, begin by locating the active copy of the session configuration file.

When you use a session configuration file in a session configuration, Windows PowerShell creates an active copy of the session configuration file and stores it in the $pshome\SessionConfig directory on the local computer.

The location of the active copy of a session configuration file is stored in the ConfigFilePath property of the session configuration object.

The following command gets the location of the session configuration file for the NoLanguage session configuration.

```
PS C:\> (Get-PSSessionConfiguration -Name NoLanguage).ConfigFilePath
C:\WINDOWS\System32\WindowsPowerShell\v1.0\SessionConfig\NoLanguage_0c115179-ff2a-4f66-a5eb-e56e5692ba22.pssc
```

You can edit the file in any text editor. The file is changed as soon as you save it and is effective in new sessions that use the session configuration.

Testing a Session Configuration File

Be sure to test all manually edited session configuration files. If the file syntax and values are not valid, users will not be able to use the session configuration to create a session.

For example, the following command tests the active session configuration file of the NoLanguage session configuration.

```
PS C:\> Test-PSSessionConfigurationFile -Path
C:\WINDOWS\System32\WindowsPowerShell\v1.0\SessionConfig\NoLanguage_0c115179-ff2a-4f66-a5eb-e56e5692ba22.pssc
```

You can use Test-PSSessionConfigurationFile to test any session configuration file, including the files the New-PSSessionConfiguration creates. For more information, see the help topic for the Test-PSSessionConfigurationFile cmdlet.

Removing a Session Configuration File

You cannot safely remove a session configuration file from a session configuration, but you can replace the file with one that has no effect.

To remove a session configuration file, create a session configuration file with the default settings and then use the Set-PSSessionConfiguration cmdlet to replace the custom session configuration file with a default version.

For example, the following command creates a Default session configuration file and then replaces the active session configuration file in the NoLanguage session configuration.

```
PS C:\> New-PSSessionConfigurationFile -Path .\Default.pssc
PS C:\> Set-PSSessionConfiguration -Name NoLanguage -Path .\Default.pssc
```

As a result of this command the NoLanguage session configuration now provides full language support (the default) in all sessions created with the session configuration.

Viewing the Properties of a Session Configuration

The session configuration objects that represent session configurations that use session configuration files have additional properties that make it easy to discover and analyze the session configuration. (Note that the type name includes a formatted view definition.)

```
PS C:\> Get-PSSessionConfiguration NoLanguage | Get-Member

   TypeName:
Microsoft.PowerShell.Commands.PSSessionConfigurationCommands#PSSessionConfiguration

Name                        MemberType     Definition
----                        ----------     ----------
Equals                      Method         bool Equals(System.Object obj)
GetHashCode                 Method         int GetHashCode()
GetType                     Method         type GetType()
ToString                    Method         string ToString()
Architecture                NoteProperty   System.String Architecture=64
Author                      NoteProperty   System.String Author=juneb
AutoRestart                 NoteProperty   System.String AutoRestart=fals
Capability                  NoteProperty   System.Object[] Capability=Sys
CompanyName                 NoteProperty   System.String CompanyName=Unkn
configfilepath              NoteProperty   System.String configfilepath=C
Copyright                   NoteProperty   System.String Copyright=(c) 20
Enabled                     NoteProperty   System.String Enabled=True
ExactMatch                  NoteProperty   System.String ExactMatch=true
ExecutionPolicy             NoteProperty   System.String ExecutionPolicy=
Filename                    NoteProperty   System.String Filename=%windir
GUID                        NoteProperty   System.String GUID=0c115179-ff
ProcessIdleTimeoutSec       NoteProperty   System.String ProcessIdleTimeo
IdleTimeoutms               NoteProperty   System.String IdleTimeoutms=72
lang                        NoteProperty   System.String lang=en-US
LanguageMode                NoteProperty   System.String LanguageMode=NoL
MaxConcurrentCommandsPerShell NoteProperty System.String MaxConcurrentCom
MaxConcurrentUsers          NoteProperty   System.String MaxConcurrentUse
MaxIdleTimeoutms            NoteProperty   System.String MaxIdleTimeoutms
MaxMemoryPerShellMB         NoteProperty   System.String MaxMemoryPerShel
MaxProcessesPerShell        NoteProperty   System.String MaxProcessesPerS
MaxShells                   NoteProperty   System.String MaxShells=300
MaxShellsPerUser            NoteProperty   System.String MaxShellsPerUser
```

```
Name                   NoteProperty   System.String Name=NoLanguage
PSVersion              NoteProperty   System.String PSVersion=3.0
ResourceUri            NoteProperty   System.String ResourceUri=http
RunAsPassword          NoteProperty   System.String RunAsPassword=
RunAsUser              NoteProperty   System.String RunAsUser=
SchemaVersion          NoteProperty   System.String SchemaVersion=1.
SDKVersion             NoteProperty   System.String SDKVersion=1
OutputBufferingMode    NoteProperty   System.String OutputBufferingM
SessionType            NoteProperty   System.String SessionType=Defa
UseSharedProcess       NoteProperty   System.String UseSharedProcess
SupportsOptions        NoteProperty   System.String SupportsOptions=
xmlns                  NoteProperty   System.String xmlns=http://sch
XmlRenderingType       NoteProperty   System.String XmlRenderingType
Permission             ScriptProperty System.Object Permission {get=
```

The new properties make it easier to search session configurations. For example, you can use the ExecutionPolicy property to find an session configuration that supports sessions with the RemoteSigned execution policy. Because the ExecutionPolicy property exists only on sessions that use session configuration files, the command might not get all qualifying session configurations.

```
PS C:\> Get-PSSessionConfiguration | where {$_.ExecutionPolicy -eq "RemoteSigned"}
```

The following command gets session configurations in which the RunAsUser is the Exchange administrator.

```
PS C:\>  Get-PSSessionConfiguration | where {$_.RunAsUser -eq "Exchange01\Admin01"}
```

NOTES

An Empty session type is designed for you to create custom sessions with selected commands. If you do not add modules, functions, or scripts to an empty session, the session is limited to expressions and might not be usable.

SEE ALSO

about_Session_Configurations
New-PSSession
Disable-PSSessionConfiguration
Enable-PSSessionConfiguration
Get-PSSessionConfiguration
New-PSSessionConfigurationFile
Register-PSSessionConfiguration
Set-PSSessionConfiguration
Test-PSSessionConfigurationFile
Unregister-PSSessionConfiguration

TOPIC

about_Signing

SHORT DESCRIPTION

Explains how to sign scripts so that they comply with the Windows PowerShell execution policies.

LONG DESCRIPTION

The Restricted execution policy does not permit any scripts to run. The AllSigned and RemoteSigned execution policies prevent Windows PowerShell from running scripts that do not have a digital signature.

This topic explains how to run selected scripts that are not signed, even while the execution policy is RemoteSigned, and how to sign scripts for your own use.

For more information about Windows PowerShell execution policies, see about_Execution_Policy.

TO PERMIT SIGNED SCRIPTS TO RUN

When you start Windows PowerShell on a computer for the first time, the Restricted execution policy (the default) is likely to be in effect.

The Restricted policy does not permit any scripts to run.

To find the effective execution policy on your computer, type:

```
Get-ExecutionPolicy
```

To run unsigned scripts that you write on your local computer and signed scripts from other users, start Windows PowerShell with the Run as Administrator option and then use the following command to change the execution policy on the computer to RemoteSigned:

Set-ExecutionPolicy RemoteSigned

For more information, see the help topic for the
Set-ExecutionPolicy cmdlet.

RUNNING UNSIGNED SCRIPTS (REMOTESIGNED EXECUTION POLICY)

If your Windows PowerShell execution policy is RemoteSigned, Windows PowerShell will not run unsigned scripts that are downloaded from the Internet, including unsigned scripts you receive through e-mail and instant messaging programs.

If you try to run a downloaded script, Windows PowerShell displays the following error message:

> The file <file-name> cannot be loaded. The file
> <file-name> is not digitally signed. The script
> will not execute on the system. Please see "Get-Help
> about_Signing" for more details.

Before you run the script, review the code to be sure that you trust it. Scripts have the same effect as any executable program.

To run an unsigned script, use the Unblock-File cmdlet or use the Following procedure.

1. Save the script file on your computer.
2. Click Start, click My Computer, and locate the saved script file.
3. Right-click the script file, and then click Properties.
4. Click Unblock.

If a script that was downloaded from the Internet is digitally signed, but you have not yet chosen to trust its publisher, Windows PowerShell displays the following message:

> Do you want to run software from this untrusted publisher?
> The file <file-name> is published by CN=<publisher-name>. This
> publisher is not trusted on your system. Only run scripts
> from trusted publishers.
>
> [V] Never run [D] Do not run [R] Run once [A] Always run
> [?] Help (default is "D"):
>
> If you trust the publisher, select "Run once" or "Always run."
> If you do not trust the publisher, select either "Never run" or
> "Do not run." If you select "Never run" or "Always run," Windows
> PowerShell will not prompt you again for this publisher.

METHODS OF SIGNING SCRIPTS

You can sign the scripts that you write and the scripts that you obtain from other sources. Before you sign any script, examine each command to verify that it is safe to run.

For best practices about code signing, see "Code-Signing Best Practices" at http://go.microsoft.com/fwlink/?LinkId=119096.

For more information about how to sign a script file, see Set-AuthenticodeSignature.

The New-SelfSignedCertificate cmdlet, introduced in the PKI module in Windows PowerShell 3.0, creates a self-signed certificate that is Appropriate for testing. For more information, see the help topic for the New-SelfSignedCertificate cmdlet.

To add a digital signature to a script, you must sign it with a code signing certificate. Two types of certificates are suitable for signing a script file:

-- Certificates that are created by a certification authority:

For a fee, a public certification authority verifies your identity and gives you a code signing certificate. When you purchase your certificate from a reputable certification authority, you are able to share your script with users on other computers that are running Windows because those other computers trust the certification authority.

-- Certificates that you create:

You can create a self-signed certificate for which your computer is the authority that creates the certificate. This certificate is free of charge and enables you to write, sign, and run scripts on your computer. However, a script signed by a self-signed certificate will not run on other computers.

Typically, you would use a self-signed certificate only to sign scripts that you write for your own use and to sign scripts that you get from other sources that you have verified to be safe. It is not appropriate for scripts that will be shared, even within an enterprise.

If you create a self-signed certificate, be sure to enable strong private key protection on your certificate. This prevents malicious

programs from signing scripts on your behalf. The instructions are included at the end of this topic.

CREATE A SELF-SIGNED CERTIFICATE

To create a self-signed certificate in use the New-SelfSignedCertificate cmdlet in the PKI module. This module is introduced in Windows PowerShell 3.0 and is included in Windows 8 and Windows Server 2012. For more information, see the help topic for the New-SelfSignedCertificate cmdlet.

To create a self-signed certificate in earlier versions of Windows, use the Certificate Creation tool (MakeCert.exe). This tool is included in the Microsoft .NET Framework SDK (versions 1.1 and later) and in the Microsoft Windows SDK.

For more information about the syntax and the parameter descriptions of the MakeCert.exe tool, see "Certificate Creation Tool (MakeCert.exe)" in the MSDN (Microsoft Developer Network) library at http://go.microsoft.com/fwlink/?LinkId=119097.

To use the MakeCert.exe tool to create a certificate, run the following commands in an SDK Command Prompt window.

Note: The first command creates a local certification authority for your computer. The second command generates a personal certificate from the certification authority.

Note: You can copy or type the commands exactly as they appear. No substitutions are necessary, although you can change the certificate name.

```
makecert -n "CN=PowerShell Local Certificate Root" -a sha1 `
   -eku 1.3.6.1.5.5.7.3.3 -r -sv root.pvk root.cer `
   -ss Root -sr localMachine

makecert -pe -n "CN=PowerShell User" -ss MY -a sha1 `
   -eku 1.3.6.1.5.5.7.3.3 -iv root.pvk -ic root.cer
```

The MakeCert.exe tool will prompt you for a private key password. The password ensures that no one can use or access the certificate without your consent. Create and enter a password that you can remember. You will use this password later to retrieve the certificate.

To verify that the certificate was generated correctly, use the following command to get the certificate in the certificate store on the computer. (You will not find a certificate file in the

file system directory.)

At the Windows PowerShell prompt, type:

```
get-childitem cert:\CurrentUser\my -codesigning
```

This command uses the Windows PowerShell Certificate provider to view information about the certificate.

If the certificate was created, the output shows the thumbprint that identifies the certificate in a display that resembles the following:

```
   Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

   Thumbprint                Subject
   ----------                -------
   4D4917CB140714BA5B81B96E0B18AAF2C4564FDF  CN=PowerShell User ]
```

SIGN A SCRIPT

After you create a self-signed certificate, you can sign scripts. If you use the AllSigned execution policy, signing a script permits you to run the script on your computer.

The following sample script, Add-Signature.ps1, signs a script. However, if you are using the AllSigned execution policy, you must sign the Add-Signature.ps1 script before you run it.

To use this script, copy the following text into a text file, and name it Add-Signature.ps1.

Note: Be sure that the script file does not have a .txt file name extension. If your text editor appends ".txt", enclose the file name in quotation marks: "add-signature.ps1".

```
## add-signature.ps1
## Signs a file
param([string] $file=$(throw "Please specify a filename."))
$cert = @(Get-ChildItem cert:\CurrentUser\My -codesigning)[0]
Set-AuthenticodeSignature $file $cert
```

To sign the Add-Signature.ps1 script file, type the following commands at the Windows PowerShell command prompt:

```
$cert = @(Get-ChildItem cert:\CurrentUser\My -codesigning)[0]

Set-AuthenticodeSignature add-signature.ps1 $cert
```

After the script is signed, you can run it on the local computer. However, the script will not run on computers on which the Windows PowerShell execution policy requires a digital signature from a trusted authority. If you try, Windows PowerShell displays the following error message:

```
The file C:\remote_file.ps1 cannot be loaded. The signature of the
certificate cannot be verified.
At line:1 char:15
+ .\ remote_file.ps1 <<<<
```

If Windows PowerShell displays this message when you run a script that you did not write, treat the file as you would treat any unsigned script. Review the code to determine whether you can trust the script.

ENABLE STRONG PRIVATE KEY PROTECTION FOR YOUR CERTIFICATE

If you have a private certificate on your computer, malicious programs might be able to sign scripts on your behalf, which authorizes Windows PowerShell to run them.

To prevent automated signing on your behalf, use Certificate Manager (Certmgr.exe) to export your signing certificate to a .pfx file. Certificate Manager is included in the Microsoft .NET Framework SDK, the Microsoft Windows SDK, and in Internet Explorer 5.0 and later versions.

To export the certificate:

1. Start Certificate Manager.

2. Select the certificate issued by PowerShell Local Certificate Root.

3. Click Export to start the Certificate Export Wizard.

4. Select "Yes, export the private key", and then click Next.

5. Select "Enable strong protection."

6. Type a password, and then type it again to confirm.

7. Type a file name that has the .pfx file name extension.

8. Click Finish.

To re-import the certificate:

1. Start Certificate Manager.

2. Click Import to start the Certificate Import Wizard.

3. Open to the location of the .pfx file that you created during the export process.

4. On the Password page, select "Enable strong private key protection", and then enter the password that you assigned during the export process.

5. Select the Personal certificate store.

6. Click Finish.

PREVENT THE SIGNATURE FROM EXPIRING

The digital signature in a script is valid until the signing certificate expires or as long as a time stamp server can verify that the script was signed while the signing certificate was valid.

Because most signing certificates are valid for one year only, using a time stamp server ensures that users can use your script for many years to come.

SEE ALSO

about_Execution_Policies
about_Profiles
Get-ExecutionPolicy
New-SelfSignedCertificate
Set-ExecutionPolicy
Set-AuthenticodeSignature
"Introduction to Code Signing" (http://go.microsoft.com/fwlink/?LinkId=106296)

TOPIC

about_Special_Characters

SHORT DESCRIPTION

Describes the special characters that you can use to control how Windows PowerShell interprets the next character in a command or parameter.

LONG DESCRIPTION

Windows PowerShell supports a set of special character sequences that are used to represent characters that are not part of the standard character set.

The special characters in Windows PowerShell begin with the backtick character, also known as the grave accent (ASCII 96).

The following special characters are recognized by Windows PowerShell:

```
`0    Null
`a    Alert
`b    Backspace
`f    Form feed
`n    New line
`r    Carriage return
`t    Horizontal tab
`v    Vertical tab
--%   Stop parsing
```

These characters are case-sensitive.

NULL (`0)

Windows PowerShell recognizes a null special character (`0) and represents it with a character code of 0. It appears as an empty space in the Windows PowerShell output. This allows you to use Windows PowerShell to read and process text files that use null characters, such as string termination or record termination indicators. The null special character

is not equivalent to the $null variable, which stores a value of NULL.

ALERT (`a)

The alert (`a) character sends a beep signal to the computer's speaker. You can use this to warn a user about an impending action. The following command sends two beep signals to the local computer's speaker:

```
for ($i = 0; $i -le 1; $i++){"`a"}
```

BACKSPACE (`b)

The backspace character (`b) moves the cursor back one character, but it does not delete any characters. The following command writes the word "backup", moves the cursor back twice, and then writes the word "out" (preceded by a space and starting at the new position):

```
"backup`b`b out"
```

The output from this command is as follows:

```
back out
```

FORM FEED (`f)

The form feed character (`f) is a print instruction that ejects the current page and continues printing on the next page. This character affects printed documents only; it does not affect screen output.

NEW LINE (`n)

The new line character (`n) inserts a line break immediately after the character.

The following example shows how to use the new line character in a Write-Host command:

```
"There are two line breaks`n`nhere."
```

The output from this command is as follows:

```
There are two line breaks

here.
```

CARRIAGE RETURN (`r)

The carriage return character (`r) eliminates the entire line prior to the `r character, as though the prior text were on a different line.

For example:

```
Write-Host "Let's not move`rDelete everything before this point."
```

The output from this command is:

```
Delete everything before this point.
```

HORIZONTAL TAB (`t)

The horizontal tab character (`t) advances to the next tab stop and continues writing at that point. By default, the Windows PowerShell console has a tab stop at every eighth space.

For example, the following command inserts two tabs between each column.

```
"Column1`t`tColumn2`t`tColumn3"
```

The output from this command is:

```
Column1        Column2        Column3
```

VERTICAL TAB (`v)

The horizontal tab character (`t) advances to the next vertical tab stop and writes all subsequent output beginning at that point. This character affects printed documents only. It does not affect screen output.

STOP PARSING (--%)

The stop-parsing symbol (--%) prevents Windows PowerShell from interpreting arguments in program calls as Windows PowerShell commands and expressions.

Place the stop-parsing symbol after the program name and before program arguments that might cause errors.

For example, the following Icacls command uses the stop-parsing symbol.

```
icacls X:\VMS --% /grant Dom\HVAdmin:(CI)(OI)F
```

Windows PowerShell sends the following command to Icacls.

```
X:\VMS /grant Dom\HVAdmin:(CI)(OI)F
```

For more information about the stop-parsing symbol, see
about_Parsing.

KEYWORDS
about_Punctuation
about_Symbols

SEE ALSO
about_Quoting_Rules
about_Escape_Characters

TOPIC

about_Splatting

SHORT DESCRIPTION
Describes how to use splatting to pass parameters to commands
in Windows PowerShell.

LONG DESCRIPTION
[This topic was contributed by Rohn Edwards of Gulfport,
Mississippi, a system administrator and the winner
of the Advanced Division of the 2012 Scripting Games.
Revised for Windows PowerShell 3.0.]

Splatting is a method of passing a collection of parameter
values to a command as unit. Windows PowerShell associates
each value in the collection with a command parameter.
Splatted parameter values are stored in named splatting
variables, which look like standard variables, but begin with
an At symbol (@) instead of a dollar sign ($). The At symbol

tells Windows PowerShell that you are passing a collection of values, instead of a single value.

Splatting makes your commands shorter and easier to read. You can re-use the splatting values in different command calls and use splatting to pass parameter values from the $PSBoundParameters automatic variable to other scripts and functions.

Beginning in Windows PowerShell 3.0, you can also use splatting to represent all parameters of a command.

SYNTAX

<CommandName> <optional parameters> @<HashTable> <optional parameters>

<CommandName> <optional parameters> @<Array> <optional parameters>

To provide parameter values for positional parameters, in which parameter names are not required, use the array syntax. To provide parameter name and value pairs, use the hash table syntax. The splatted value can appear anywhere in the parameter list.

When splatting, you do not need to use a hash table or an array to pass all parameters. You may pass some parameters by using splatting and pass others by position or by parameter name. Also, you can splat multiple objects in a single command just so you pass no more than one value for each parameter.

SPLATTING WITH HASH TABLES

Use a hash table to splat parameter name and value pairs. You can use this format for all parameter types, including positional and named parameters and switch parameters.

The following examples compare two Copy-Item commands that copy the Test.txt file to the Test2.txt file in the same directory.

The first example uses the traditional format in which parameter names are included.

```
Copy-Item -Path "test.txt" -Destination "test2.txt" -WhatIf
```

The second example uses hash table splatting. The first command creates a hash table of parameter-name and parameter-value pairs and stores it in the $HashArguments variable. The second command uses the $HashArguments variable in a command with splatting. The At symbol (@HashArguments)

replaces the dollar sign ($HashArguments) in the command.

To provide a value for the WhatIf switch parameter, use $True or $False.

```
PS C:\>$HashArguments = @{ Path = "test.txt"; Destination = "test2.txt"; WhatIf = $true }
PS C:\>Copy-Item @HashArguments
```

Note: In the first command, the At symbol (@) indicates a hash table, not a splatted value. The syntax for hash tables in Windows PowerShell is: @{ <name>=<value>; <name>=<value>; …}

SPLATTING WITH ARRAYS

Use an array to splat values for positional parameters, which do not require parameter names. The values must be in position-number order in the array.

The following examples compare two Copy-Item commands that copy the Test.txt file to the Test2.txt file in the same directory.

The first example uses the traditional format in which parameter names are omitted. The parameter values appear in position order in the command.

```
Copy-Item "test.txt" "test2.txt" -WhatIf
```

The second example uses array splatting. The first command creates an array of the parameter values and stores it in the $ArrayArguments variable. The values are in position order in the array. The second command uses the $ArrayArguments variable in a command in splatting. The At symbol (@ArrayArguments) replaces the dollar sign ($ArrayArguments) in the command.

```
PS C:\>$ArrayArguments = "test.txt", "test2.txt"
PS C:\>Copy-Item @ArrayArguments -WhatIf
```

EXAMPLES

This example shows how to re-use splatted values in different commands. The commands in this example use the Write-Host cmdlet to write messages to the host program console. It uses splatting to specify the foreground and background colors.

To change the colors of all commands, just change the value of the $Colors variable.

The first command creates a hash table of parameter names and values and stores the hash table in the $Colors variable.

```
$Colors = @{ForegroundColor = "black"
        BackgroundColor = "white"}
```

The second and third commands use the $Colors variable for splatting in a Write-Host command. To use the $Colors variable, replace the dollar sign ($Colors) with an At symbol (@Colors).

```
# Write a message with the colors in $Colors
Write-Host "This is a test." @Colors

# Write second message with same colors.
# The position of splatted hash table does not matter.
Write-Host @Colors "This is another test."
```

This example shows how to forward their parameters to other commands by using splatting and the $PSBoundParameters automatic variable.

The $PSBoundParameters automatic variable is a dictionary (System.Collections.Generic.Dictionary) that contains all of the parameter names and values that are used when a script or function is run.

In the following example, we use the $PSBoundParameters variable to forward the parameters values passed to a script or function from Test2 function to the Test1 function. Both calls to the Test1 function from Test2 use splatting.

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

   $a
   $b
   $c
}

function Test2
{
   param($a, $b, $c)

   # Call the Test1 function with $a, $b, and $c.
   Test1 @PsBoundParameters

   # Call the Test1 function with $b and $c, but not with $a
   $LimitedParameters = $PSBoundParameters
   $LimitedParameters.Remove("a") | Out-Null
   Test1 @LimitedParameters
}

PS C:\> Test2 -a 1 -b 2 -c 3
```

1
2
3
2
3

SPLATTING COMMAND PARAMETERS

You can use splatting to represent the parameters of a command. This technique is useful when you are creating a proxy function, that is, a function that calls another command. This feature is introduced in Windows PowerShell 3.0.

To splat the parameters of a command, use @Args to represent the command parameters. This technique is easier than enumerating command parameters and it works without revision even if the parameters of the called command change.

The feature uses the $Args automatic variable, which contains all unassigned parameter values.

For example, the following function calls the Get-Process cmdlet. In this function, @Args represents all of the parameters of the Get-Process cmdlet.

```
function Get-MyProcess { Get-Process @Args }
```

When you use the Get-MyProcess function, all unassigned parameters and parameter values are passed to @Args, as shown in the following commands.

```
PS C:\> Get-MyProcess -Name PowerShell

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    463     46   225484     237196   719    15.86   3228 powershell


PS C:\> Get-MyProcess -Name PowerShell_Ise -FileVersionInfo

ProductVersion   FileVersion      FileName
--------------   -----------      --------
6.2.9200.16384   6.2.9200.1638...
C:\Windows\system32\WindowsPowerShell\v1.0\PowerShell_ISE.exe
```

You can use @Args in a function that has explicitly declared parameters. You can use it more than once in

a function, but all parameters that you enter are passed
to all instances of @Args, as shown in the following
example.

```
function Get-MyCommand
{
   Param ([switch]$P, [switch]$C)
   if ($P) { Get-Process @Args }
   if ($C) { Get-Command @Args }
}

PS C:\> Get-MyCommand -P -C -Name PowerShell

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    408     28    75568     83176   620     1.33   1692 powershell

Path            : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Extension       : .exe
Definition      : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Visibility      : Public
OutputType      : {System.String}
Name            : powershell.exe
CommandType     : Application
ModuleName      :
Module          :
RemotingCapability : PowerShell
Parameters      :
ParameterSets   :
HelpUri         :
FileVersionInfo : File:           C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
```

SEE ALSO

about_Arrays
about_Automatic_Variables
about_Hash_Tables
about_Parameters

TOPIC

about_Split

SHORT DESCRIPTION

Explains how to use the Split operator to split one or more strings into substrings.

LONG DESCRIPTION

The Split operator splits one or more strings into substrings. You can change the following elements of the Split operation:

-- Delimiter. The default is whitespace, but you can specify characters, strings, patterns, or script blocks that specify the delimiter. The Split operator in Windows PowerShell uses a regular expression in the delimiter, rather than a simple character.

-- Maximum number of substrings. The default is to return all substrings. If you specify a number less than the number of substrings, the remaining substrings are concatenated in the last substring.

-- Options that specify the conditions under which the delimiter is matched, such as SimpleMatch and Multiline.

SYNTAX

The following diagram shows the syntax for the -split operator.

The parameter names do not appear in the command. Include only the parameter values. The values must appear in the order specified in the syntax diagram.

-Split <String>

<String> -Split <Delimiter>[,<Max-substrings>[,"<Options>"]]

<String> -Split {<ScriptBlock>} [,<Max-substrings>]

You can substitute -iSplit or -cSplit for -split in any binary Split statement

(a Split statement that includes a delimiter or script block). The -iSplit and -split operators are case-insensitive. The -cSplit operator is case-sensitive, meaning that case is considered when the delimiter rules are applied.

PARAMETERS

<String>
Specifies one or more strings to be split. If you submit multiple strings, all the strings are split using the same delimiter rules. Example:

```
-split "red yellow blue green"
red
    yellow
blue
green
```

<Delimiter>
The characters that identify the end of a substring. The default delimiter is whitespace, including spaces and non-printable characters, such as newline (`n) and tab (`t). When the strings are split, the delimiter is omitted from all the substrings. Example:

```
"Lastname:FirstName:Address" -split ":"
Lastname
FirstName
Address
```

By default, the delimiter is omitted from the results. To preserve all or part of the delimiter, enclose in parentheses the part that you want to preserve. If the <Max-substrings> parameter is added, this takes precedence when your command splits up the collection. If you opt to include a delimiter as part of the output, the command returns the delimiter as part of the output; however, splitting the string to return the delimiter as part of output does not count as a split. .Examples:

```
    "Lastname:FirstName:Address" -split "(:)"
Lastname
    :
FirstName
:
    Address


    "Lastname/:/FirstName/:/Address" -split "/(:)/"
Lastname
    :
```

```
FirstName
    :
    Address
```

In the following example, <Max-substrings> is set to 3. This results in three splits of the string values, but a total of five strings in the resulting output; the delimiter is included after the splits, until the maximum of three substrings is reached. Additional delimiters in the final substring become part of the substring.

```
'Chocolate-Vanilla-Strawberry-Blueberry' -split '(-)', 3;

Chocolate
-
Vanilla
-
Strawberry-Blueberry
```

<Max-substrings>

Specifies the maximum number of times that a string is split. The default is all the substrings split by the delimiter. If there are more substrings, they are concatenated to the final substring. If there are fewer substrings, all the substrings are returned. A value of 0 and negative values return all the substrings.

Max-substrings does not specify the maximum number of objects that are returned; its value equals the maximum number of times that a string is split.
If you submit more than one string (an array of strings) to the Split operator , the Max-substrings limit is applied to each string separately. Example:

```
$c = "Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,Neptune"
$c -split ",", 5
Mercury
Venus
Earth
Mars
Jupiter,Saturn,Uranus,Neptune
```

<ScriptBlock>

An expression that specifies rules for applying the delimiter. The expression must evaluate to $true or $false. Enclose the script block in braces. Example:

```
    $c = "Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,Neptune"
$c -split {$_ -eq "e" -or $_ -eq "p"}
M
```

rcury,V
nus,Earth,Mars,Ju
it
r,Saturn,Uranus,N

tun

<Options>
Enclose the option name in quotation marks. Options are valid only when the <Max-substrings> parameter is used in the statement.

The syntax for the Options parameter is:

"SimpleMatch [,IgnoreCase]"

"[RegexMatch] [,IgnoreCase] [,CultureInvariant] [,IgnorePatternWhitespace] [,ExplicitCapture] [,Singleline | ,Multiline]"

The SimpleMatch options are:

-- SimpleMatch: Use simple string comparison when evaluating the delimiter. Cannot be used with RegexMatch.

-- IgnoreCase: Forces case-insensitive matching, even if the -cSplit operator is specified.

The RegexMatch options are:

-- RegexMatch: Use regular expression matching to evaluate the delimiter. This is the default behavior. Cannot be used with SimpleMatch.

-- IgnoreCase: Forces case-insensitive matching, even if the -cSplit operator is specified.

-- CultureInvariant: Ignores cultural differences in language when evaluting the delimiter. Valid only with RegexMatch.

-- IgnorePatternWhitespace: Ignores unescaped whitespace and comments marked with the number sign (#). Valid only with RegexMatch.

-- Multiline: Multiline mode recognizes the start and end of lines and strings. Valid only with RegexMatch. Singleline is the default.

-- Singleline: Singleline mode recognizes only the start and end of strings. Valid only with RegexMatch. Singleline is the default.

-- ExplicitCapture: Ignores non-named match groups so that only explicit capture groups are returned in the result list. Valid only with RegexMatch.

UNARY and BINARY SPLIT OPERATORS

The unary split operator (-split <string>) has higher precedence than a comma. As a result, if you submit a comma-separated list of strings to the unary split operator, only the first string (before the first comma) is split.

To split more than one string, use the binary split operator (<string> -split <delimiter>). Enclose all the strings in parentheses, or store the strings in a variable, and then submit the variable to the split operator.

Consider the following example:

```
-split "1 2", "a b"
1
2
a b


"1 2", "a b" -split " "
1
2
a
b


-split ("1 2", "a b")
1
2
a
b

$a = "1 2", "a b"
-split $a
1
2
a
b
```

EXAMPLES

The following statement splits the string at whitespace.

```
C:\PS> -split "Windows PowerShell 2.0`nWindows PowerShell with remoting"

Windows
PowerShell
2.0
Windows
PowerShell
with
remoting
```

The following statement splits the string at any comma.

```
C:\PS> "Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,Neptune" -split ','

    Mercury
    Venus
    Earth
    Mars
    Jupiter
    Saturn
    Uranus
    Neptune
```

The following statement splits the string at the pattern "er".

```
C:\PS>"Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,Neptune" -split 'er'

M
cury,Venus,Earth,Mars,Jupit
,Saturn,Uranus,Neptune
```

The following statement performs a case-sensitive split at the letter "N".

```
C:\PS> "Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,Neptune" -cSplit 'N'

Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,
eptune
```

The following statement splits the string at "e" and "t".

```
C:\PS> "Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,Neptune" -split '[et]'

M
rcury,V
nus,
ar
h,Mars,Jupi

r,Sa
urn,Uranus,N
p
un
```

The following statement splits the string at "e" and "r", but limits the resulting substrings to six substrings.

```
C:\PS> "Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,Neptune" -split '[er]', 6

M

cu
y,V
nus,
arth,Mars,Jupiter,Saturn,Uranus,Neptune
```

The following statement splits a string into three substrings.

```
C:\PS> "a,b,c,d,e,f,g,h" -split ",", 3

a
b
c,d,e,f,g,h
```

The following statement splits two strings into three substrings. (The limit is applied to each string independently.)

```
C:\PS> "a,b,c,d", "e,f,g,h" -split ",", 3

a
b
```

c,d
e
f
g,h

The following statement splits each line in the here-string at the first digit. It uses the Multiline option to recognize the beginning of each line and string.

The 0 represents the "return all" value of the Max-substrings parameter. You can use options, such as Multiline, only when the Max-substrings value is specified.

```
C:\PS> $a = @'
1The first line.
2The second line.
3The third of three lines.
'@

C:\PS> $a -split "^\d", 0, "multiline"

The first line.


The second line.


The third of three lines.
```

The following statement uses the SimpleMatch option to direct the -split operator to interpret the dot (.) delimiter literally.

With the default, RegexMatch, the dot enclosed in quotation marks (".") is interpreted to match any character except for a newline character. As a result, the Split statement returns a blank line for every character except newline.

The 0 represents the "return all" value of the Max-substrings parameter. You can use options, such as SimpleMatch, only when the Max-substrings value is specified.

```
C:\PS> "This.is.a.test" -split ".", 0, "simplematch"

This
is
```

a
test

The following statement splits the string at one of two delimiters, depending on the value of a variable.

```
C:\PS> $i = 1
C:\PS> $c = "LastName, FirstName; Address, City, State, Zip"
C:\PS> $c -split {if ($i -lt 1) {$_ -eq ","} else {$_ -eq ";"}}
```

```
LastName, FirstName
 Address, City, State, Zip
```

The following split statements split an XML file first at the angle bracket and then at the semicolon. The result is a readable version of the XML file.

```
C:\PS> get-process PowerShell | export-clixml ps.xml
C:\PS> $x = import-clixml ps.xml
C:\PS> $x = $x -split "<"
C:\PS> $x = $x -split ";"
```

To display the result, type "$x".

```
C:\PS> $x

    @{__NounName=Process
Name=PowerShell
Handles=428
VM=150081536
WS=34840576
PM=36253696
...
```

SEE ALSO

Split-Path
about_Operators
about_Comparison_Operators
about_Join

```
Name                    Category  Module              Synopsis
----                    --------  ------              --------
about_Suspend-Workflow  HelpFile                      Describes the Suspend-Workflow activity, which
suspends
about_Suspend-Workflow  HelpFile                      Describes the Suspend-Workflow activity, which
suspends
```

TOPIC

about_Switch

SHORT DESCRIPTION

Explains how to use a switch to handle multiple If statements.

LONG DESCRIPTION

To check a condition in a script or function, use an If statement. The If can check many types of conditions, including the value of variables and the properties of objects.

To check multiple conditions, use a Switch statement. The Switch statement is equivalent to a series of If statements, but it is simpler. The Switch statement lists each condition and an optional action. If a condition obtains, the action is performed.

A basic Switch statement has the following format:

```
Switch (<test-value>)
{
   <condition> {<action>}
   <condition> {<action>}
}
```

For example, the following Switch statement compares the test

value, 3, to each of the conditions. When the test value matches the condition, the action is performed.

```
PS> switch (3)
{
  1 {"It is one."}
  2 {"It is two."}
  3 {"It is three."}
  4 {"It is four."}
}
It is three.
```

In this simple example, the value is compared to each condition in the list, even though there is a match for the value 3. The following Switch statement has two conditions for a value of 3. It demonstrates that, by default, all conditions are tested.

```
PS> switch (3)
{
  1 {"It is one."}
  2 {"It is two."}
  3 {"It is three."}
  4 {"It is four."}
  3 {"Three again."}
}
It is three.
Three again.
```

To direct the Switch to stop comparing after a match, use the Break statement. The Break statement terminates the Switch statement.

```
PS> switch (3)
{
  1 {"It is one."}
  2 {"It is two."}
  3 {"It is three."; Break}
  4 {"It is four."}
  3 {"Three again."}
}
It is three.
```

If the test value is a collection, such as an array, each item in the collection is evaluated in the order in which it appears. The following examples evaluates 4 and then 2.

```
PS> switch (4, 2)
{
```

```
    1 {"It is one." }
    2 {"It is two." }
    3 {"It is three." }
    4 {"It is four." }
    3 {"Three again."}
  }
 It is four.
 It is two.
```

Any Break statements apply to the collection, not to each value, as shown in the following example. The Switch statement is terminated by the Break statement in the condition of value 4.

```
 PS> switch (4, 2)
 {
   1 {"It is one."; Break}
   2 {"It is two." ; Break }
   3 {"It is three." ; Break }
   4 {"It is four." ; Break }
   3 {"Three again."}
 }
It is four.
```

SYNTAX

The complete Switch statement syntax is as follows:

```
switch [-regex|-wildcard|-exact][-casesensitive] (<value>)
```

or

```
switch [-regex|-wildcard|-exact][-casesensitive] -file filename
```

followed by

```
{
  "string"|number|variable|{ expression } { statementlist }
  default { statementlist }
}
```

If no parameters are used, Switch performs a case-insensitive exact match for the value. If the value is a collection, each element is evaluated in the order in which it appears.

The Switch statement must include at least one condition statement.

The Default clause is triggered when the value does not match

any of the conditions. It is equivalent to an Else clause in an If statement. Only one Default clause is permitted in each Switch statement.

Switch has the following parameters:

Regex Performs regular expression matching of the value to the condition. If you use Regex, Wildcard and Exact are ignored. Also, if the match clause is not a string, this parameter is ignored.

```
Example:
PS> switch ("fourteen")
  {
    1 {"It is one."; Break}
    2 {"It is two."; Break}
    3 {"It is three."; Break}
    4 {"It is four."; Break}
    3 {"Three again."; Break}
    "fo*" {"That's too many."}
  }

PS> switch -Regex ("fourteen")
  {
    1 {"It is one."; Break}
    2 {"It is two."; Break}
    3 {"It is three."; Break}
    4 {"It is four."; Break}
    3 {"Three again."; Break}
    "fo*" {"That's too many."}
  }
That's too many.
```

Wildcard Indicates that the condition is a wildcard string. If you use Wildcard, Regex and Exact are ignored. Also, if the match clause is not a string, this parameter is ignored.

Exact Indicates that the match clause, if it is a string, must match exactly. If you use Exact, Regex and Wildcard and Exact are ignored. Also, if the match clause is not a string, this parameter is ignored.

CaseSensitivePerforms a case-sensitive match. If the match clause is not a string, this parameter is ignored.

File Takes input from a file rather than a value

statement. If multiple File parameters are
included, only the last one is used. Each line of the
file is read and evaluated by the Switch statement.

Multiple instances of Regex, Wildcard, or Exact are permitted. However, only the last parameter used is effective.

If the value matches multiple conditions, the action for each condition is executed. To change this behavior, use the Break or Continue keywords.

The Break keyword stops processing and exits the Switch statement.

The Continue keyword continues processing the current value and any subsequent values.

If the condition is an expression or a script block, it is evaluated just before it is compared to the value. The value is assigned to the $_ automatic variable and is available in the expression. The match succeeds if the expression is true or matches the value. The expression is evaluated in its own scope.

The "Default" keyword specifies a condition that is evaluated only when no other conditions match the value.

The action for each condition is independent of the actions in other conditions. The closing brace (}) in the action is an explicit break.

SEE ALSO

about_Break
about_Continue
about_If
about_Script_Blocks

TOPIC

about_Throw

SHORT DESCRIPTION

Describes the Throw keyword, which generates a terminating error.

LONG DESCRIPTION

The Throw keyword causes a terminating error. You can use the Throw keyword to stop the processing of a command, function, or script.

For example, you can use the Throw keyword in the script block of an If statement to respond to a condition or in the Catch block of a Try-Catch-Finally statement. You can also use the Throw keyword in a parameter declaration to make a function parameter mandatory.

The Throw keyword can throw any object, such as a user message string or the object that caused the error.

SYNTAX

The syntax of the Throw keyword is as follows:

```
throw [<expression>]
```

The expression in the Throw syntax is optional. When the Throw statement does not appear in a Catch block, and it does not include an expression, it generates a ScriptHalted error.

```
C:\PS> throw

ScriptHalted
At line:1 char:6
+ throw <<<<
    + CategoryInfo          : OperationStopped: (:) [], RuntimeException
    + FullyQualifiedErrorId : ScriptHalted
```

If the Throw keyword is used in a Catch block without an expression, it throws the current RuntimeException again. For more information, see about_Try_Catch_Finally.

THROWING A STRING

The optional expression in a Throw statement can be a string, as shown in the following example:

```
C:\PS> throw "This is an error."

This is an error.
At line:1 char:6
+ throw <<<<  "This is an error."
   + CategoryInfo          : OperationStopped: (This is an error.:String) [], RuntimeException
   + FullyQualifiedErrorId : This is an error.
```

THROWING OTHER OBJECTS

The expression can also be an object that throws the object that represents the PowerShell process, as shown in the following example:

```
C:\PS> throw (get-process PowerShell)

System.Diagnostics.Process (PowerShell)
At line:1 char:6
+ throw <<<<  (get-process PowerShell)
   + CategoryInfo          : OperationStopped: (System.Diagnostics.Process (PowerShell):Process) [],
RuntimeException
   + FullyQualifiedErrorId : System.Diagnostics.Process (PowerShell)
```

You can use the TargetObject property of the ErrorRecord object in the $error automatic variable to examine the error.

```
C:\PS> $error[0].targetobject

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    319     26    61016      70864   568     3.28   5548 PowerShell
```

You can also throw an ErrorRecord object or a Microsoft .NET Framework exception. The following example uses the Throw keyword to throw a System.FormatException object.

```
C:\PS> $formatError = new-object system.formatexception

C:\PS> throw $formatError

One of the identified items was in an invalid format.
At line:1 char:6
+ throw <<<<  $formatError
   + CategoryInfo          : OperationStopped: (:) [], FormatException
   + FullyQualifiedErrorId : One of the identified items was in an invalid format.
```

RESULTING ERROR

The Throw keyword can generate an ErrorRecord object. The Exception property of the ErrorRecord object contains a RuntimeException object. The remainder of the ErrorRecord object and the RuntimeException object vary with the object that the Throw keyword throws.

The RunTimeException object is wrapped in an ErrorRecord object, and the ErrorRecord object is automatically saved in the $Error automatic variable.

USING THROW TO CREATE A MANDATORY PARAMETER

You can use the Throw keyword to make a function parameter mandatory.

This is an alternative to using the Mandatory parameter of the Parameter keyword. When you use the Mandatory parameter, the system prompts the user for the required parameter value. When you use the Throw keyword, the command stops and displays the error record.

For example, the Throw keyword in the parameter subexpression makes the Path parameter a required parameter in the function.

In this case, the Throw keyword throws a message string, but it is the presence of the Throw keyword that generates the terminating error if the Path parameter is not specified. The expression that follows Throw is optional.

```
function Get-XMLFiles
{
   param ($path = $(throw "The Path parameter is required."))
   dir -path $path\*.xml -recurse | sort lastwritetime | ft lastwritetime, attributes, name  -auto
}
```

SEE ALSO

about_Break
about_Continue
about_Scope
about_Trap
about_Try_Catch_Finally

TOPIC

about_Transactions

SHORT DESCRIPTION

Describes how to manage transacted operations in Windows PowerShell.

LONG DESCRIPTION

Transactions are supported in Windows PowerShell beginning in Windows PowerShell 2.0. This feature enables you to start a transaction, to indicate which commands are part of the transaction, and to commit or roll back a transaction.

ABOUT TRANSACTIONS

In Windows PowerShell, a transaction is a set of one or more commands that are managed as a logical unit. A transaction can be completed ("committed"), which changes data affected by the transaction. Or, a transaction can be completely undone ("rolled back") so that the affected data is not changed by the transaction.

Because the commands in a transaction are managed as a unit, either all commands are committed, or all commands are rolled back.

Transactions are widely used in data processing, most notably in database operations and for financial transactions. Transactions are most often used when the worst-case scenario for a set of commands is not that they all fail, but that some commands succeed while others fail, leaving the system in a damaged, false, or uninterpretable state that is difficult to repair.

TRANSACTION CMDLETS

Windows PowerShell includes several cmdlets designed for managing transactions.

Cmdlet	Description

---------- ----------

Start-Transaction Starts a new transaction.

Use-Transaction Adds a command or expression to the transaction. The command must use transaction-enabled objects.

Undo-Transaction Rolls back the transaction so that no data is changed by the transaction.

Complete-Transaction Commits the transaction. The data affected by the transaction is changed.

Get-Transaction Gets information about the active transaction.

For a list of transaction cmdlets, type:

```
get-command *transaction
```

For detailed information about the cmdlets, type:

```
get-help <cmdlet-name> -detailed
```

For example:

```
get-help use-transaction -detailed
```

TRANSACTION-ENABLED ELEMENTS

To participate in a transaction, both the cmdlet and the provider must support transactions. This feature is built in to the objects that are affected by the transaction.

The Windows PowerShell Registry provider supports transactions in Windows Vista. The TransactedString object (Microsoft.PowerShell.Commands.Management.TransactedString) works with any operating system that runs Windows PowerShell.

Other Windows PowerShell providers can support transactions. To find the Windows PowerShell providers in your session that support transactions, use the following command to find the "Transactions" value in the Capabilities property of providers:

```
get-psprovider | where {$_.Capabilities -like "*transactions*"}
```

For more information about a provider, see the Help for the provider.
To get provider Help, type:

get-help <provider-name>

For example, to get Help for the Registry provider, type:

get-help registry

THE USETRANSACTION PARAMETER

Cmdlets that can support transactions have a UseTransaction parameter. This parameter includes the command in the active transaction. You can use the full parameter name or its alias, "usetx".

The parameter can be used only when the session contains an active transaction. If you enter a command with the UseTransaction parameter when there is no active transaction, the command fails.

To find cmdlets with the UseTransaction parameter, type:

get-help * -parameter UseTransaction

In Windows PowerShell core, all of the cmdlets designed to work with Windows PowerShell providers support transactions. As a result, you can use the provider cmdlets to manage transactions.

For more information about Windows PowerShell providers, see about_Providers.

THE TRANSACTION OBJECT

Transactions are represented in Windows PowerShell by a transaction object, System.Management.Automation.Transaction.

The object has the following properties:

RollbackPreference:
Contains the rollback preference set for the current transaction. You can set the rollback preference when you use Start-Transaction to start the transaction.

The rollback preference determines the conditions under which the transaction is rolled back automatically. Valid

values are Error, TerminatingError, and Never. The default value is Error.

Status:
Contains the current status of the transaction. Valid values are Active, Committed, and RolledBack.

SubscriberCount:
Contains the number of subscribers to the transaction. A subscriber is added to a transaction when you start a transaction while another transaction is in progress. The subscriber count is decremented when a subscriber commits the transaction.

ACTIVE TRANSACTIONS

In Windows PowerShell, only one transaction is active at a time, and you can manage only the active transaction. Multiple transactions can be in progress in the same session at the same time, but only the most-recently started transaction is active.

As a result, you cannot specify a particular transaction when using the transaction cmdlets. Commands always apply to the active transaction.

This is most evident in the behavior of the Get-Transaction cmdlet. When you enter a Get-Transaction command, Get-Transaction always gets only one transaction object. This object is the object that represents the active transaction.

To manage a different transaction, you must first finish the active transaction, either by committing it or rolling it back. When you do this, the previous transaction becomes active automatically. Transactions become active in the reverse of order of which they are started, so that the most recently started transaction is always active.

SUBSCRIBERS AND INDEPENDENT TRANSACTIONS

If you start a transaction while another transaction is in progress, by default, Windows PowerShell does not start a new transaction. Instead, it adds a "subscriber" to the current transaction.

When a transaction has multiple subscribers, a single Undo-Transaction command at any point rolls back the entire

transaction for all subscribers. However, to commit the transaction, you must enter a Complete-Transaction command for every subscriber.

To find the number of subscribers to a transaction, check the SubscriberCount property of the transaction object. For example, the following command uses the Get-Transaction cmdlet to get the value of the SubscriberCount property of the active transaction:

```
(Get-Transaction).SubscriberCount
```

Adding a subscriber is the default behavior because most transactions that are started while another transaction is in progress are related to the original transaction. In the typical model, a script that contains a transaction calls a helper script that contains its own transaction. Because the transactions are related, they should be rolled back or committed as a unit.

However, you can start a transaction that is independent of the current transaction by using the Independent parameter of the Start-Transaction cmdlet.

When you start an independent transaction, Start-Transaction creates a new transaction object, and the new transaction becomes the active transaction. The independent transaction can be committed or rolled back without affecting the original transaction.

When the independent transaction is finished (committed or rolled back), the original transaction becomes the active transaction again.

CHANGING DATA

When you use transactions to change data, the data that is affected by the transaction is not changed until you commit the transaction. However, the same data can be changed by commands that are not part of the transaction.

Keep this in mind when you are using transactions to manage shared data. Typically, databases have mechanisms that lock the data while you are working on it, preventing other users, and other commands, scripts, and functions, from changing it.

However, the lock is a feature of the database. It is not related to transactions. If you are working in a transaction-enabled file system or other data store, the data can be changed while the transaction is in progress.

EXAMPLES

The examples in this section use the Windows PowerShell Registry provider and assume that you are familiar with it. For information about the Registry provider, type "get-help registry".

EXAMPLE 1: COMMITTING A TRANSACTION

To create a transaction, use the Start-Transaction cmdlet. The following command starts a transaction with the default settings.

```
start-transaction
```

To include commands in the transaction, use the UseTransaction parameter of the cmdlet. By default, commands are not included in the transaction,

For example, the following command, which sets the current location in the Software key of the HKCU: drive, is not included in the transaction.

```
cd hkcu:\Software
```

The following command, which creates the MyCompany key, uses the UseTransaction parameter of the New-Item cmdlet to include the command in the active transaction.

```
new-item MyCompany -UseTransaction
```

The command returns an object that represents the new key, but because the command is part of the transaction, the registry is not yet changed.

```
Hive: HKEY_CURRENT_USER\Software

SKC  VC Name                           Property
---  -- ----                           --------
  0   0 MyCompany                       {}
```

To commit the transaction, use the Complete-Transaction cmdlet. Because it always affects the active transaction, you cannot specify the transaction.

```
complete-transaction
```

As a result, the MyCompany key is added to the registry.

```
dir m*

    Hive: HKEY_CURRENT_USER\software

SKC  VC Name                           Property
---  -- ----                       --------
 83   1 Microsoft                  {(default)}
  0   0 MyCompany                    {}
```

EXAMPLE 2: ROLLING BACK A TRANSACTION

To create a transaction, use the Start-Transaction cmdlet. The following command starts a transaction with the default settings.

```
start-transaction
```

The following command, which creates the MyOtherCompany key, uses the UseTransaction parameter of the New-Item cmdlet to include the command in the active transaction.

```
new-item MyOtherCompany -UseTransaction
```

The command returns an object that represents the new key, but because the command is part of the transaction, the registry is not yet changed.

```
    Hive: HKEY_CURRENT_USER\Software

SKC  VC Name                           Property
---  -- ----                       --------
  0   0 MyOtherCompany               {}
```

To roll back the transaction, use the Undo-Transaction cmdlet. Because it always affects the active transaction, you do not specify the transaction.

```
Undo-transaction
```

The result is that the MyOtherCompany key is not added to the registry.

```
dir m*

    Hive: HKEY_CURRENT_USER\software
```

```
SKC  VC Name                   Property
---  -- ----               --------
 83   1 Microsoft              {(default)}
  0   0 MyCompany                {}
```

EXAMPLE 3: PREVIEWING A TRANSACTION

Typically, the commands used in a transaction change data. However, the commands that get data are useful in a transaction, too, because they get data inside of the transaction. This provides a preview of the changes that committing the transaction would cause.

The following example shows how to use the Get-ChildItem command (the alias is "dir") to preview the changes in a transaction.

The following command starts a transaction.

```
start-transaction
```

The following command uses the New-ItemProperty cmdlet to add the MyKey registry entry to the MyCompany key. The command uses the UseTransaction parameter to include the command in the transaction.

```
new-itemproperty -path MyCompany -Name MyKey -value 123 -UseTransaction
```

The command returns an object representing the new registry entry, but the registry entry is not changed.

```
MyKey
-----
123
```

To get the items that are currently in the registry, use a Get-ChildItem command ("dir") without the UseTransaction parameter. The following command gets items that begin with "M."

```
dir m*
```

The result shows that no entries have yet been added to the MyCompany key.

```
Hive: HKEY_CURRENT_USER\Software
```

```
SKC  VC Name                   Property
---  -- ----                   --------
83   1 Microsoft                {(default)}
 0   0 MyCompany                  {}
```

To preview the effect of committing the transaction, enter a Get-ChildItem ("dir") command with the UseTransaction parameter. This command has a view of the data from within the transaction.

```
dir m* -useTransaction
```

The result shows that, if the transaction is committed, the MyKey entry will be added to the MyCompany key.

```
Hive: HKEY_CURRENT_USER\Software

SKC  VC Name                   Property
---  -- ----                   --------
83   1 Microsoft                {(default)}
 0   1 MyCompany                  {MyKey}
```

EXAMPLE 4: COMBINING TRANSACTED AND NON-TRANSACTED COMMANDS

You can enter non-transacted commands during a transaction. The non-transacted commands affect the data immediately, but they do not affect the transaction.

The following command starts a transaction in the HKCU:\Software registry key.

```
start-transaction
```

The next three commands use the New-Item cmdlet to add keys to the registry. The first and third commands use the UseTransaction parameter to include the commands in the transaction. The second command omits the parameter. Because the second command is not included in the transaction, it is effective immediately.

```
new-item MyCompany1 -UseTransaction

new-item MyCompany2
```

```
new-item MyCompany3 -UseTransaction
```

To view the current state of the registry, use a Get-ChildItem ("dir") command without the UseTransaction parameter. This command gets items that begin with "M."

```
dir m*
```

The result shows that the MyCompany2 key is added to the registry, but the MyCompany1 and MyCompany3 keys, which are part of the transaction, are not added.

```
Hive: HKEY_CURRENT_USER\Software

SKC  VC Name                   Property
---  -- ----                   --------
83   1 Microsoft              {(default)}
0    0 MyCompany2               {}
```

The following command commits the transaction.

```
complete-transaction
```

Now, the keys that were added as part of the transaction appear in the registry.

```
dir m*


    Hive: HKEY_CURRENT_USER\Software

SKC  VC Name                   Property
---  -- ----                   --------
83   1 Microsoft              {(default)}
0    0 MyCompany1               {}
0    0 MyCompany2               {}
0    0 MyCompany3               {}
```

EXAMPLE 5: USING AUTOMATIC ROLLBACK

When a command in a transaction generates an error of any kind, the transaction is automatically rolled back.

This default behavior is designed for scripts that run transactions. Scripts are typically well tested and include error-handling logic, so errors are not expected and should terminate the transaction.

The first command starts a transaction in the HKCU:\Software registry key.

```
start-transaction
```

The following command uses the New-Item cmdlet to add the MyCompany key to the registry. The command uses the UseTransaction parameter (the alias is "usetx") to include the command in the transaction.

```
New-Item MyCompany -UseTX
```

Because the MyCompany key already exists in the registry, the command fails, and the transaction is rolled back.

```
New-Item : A key at this path already exists
At line:1 char:9
+ new-item <<<<  MyCompany -usetx
```

A Get-Transaction command confirms that the transaction has been rolled back and that the SubscriberCount is 0.

```
RollbackPreference   SubscriberCount   Status
------------------   ---------------   ------
Error                0                 RolledBack
```

EXAMPLE 6: CHANGING THE ROLLBACK PREFERENCE

If you want the transaction to be more error tolerant, you can use the RollbackPreference parameter of Start-Transaction to change the preference.

The following command starts a transaction with a rollback preference of "Never".

```
start-transaction -rollbackpreference Never
```

In this case, when the command fails, the transaction is not automatically rolled back.

```
New-Item MyCompany -UseTX
```

```
    New-Item : A key at this path already exists
    At line:1 char:9
    + new-item <<<<  MyCompany -usetx
```

Because the transaction is still active, you can
resubmit the command as part of the transaction.

```
New-Item MyOtherCompany -UseTX
```

EXAMPLE 7: USING THE USE-TRANSACTION CMDLET

The Use-Transaction cmdlet enables you to do direct scripting
against transaction-enabled Microsoft .NET Framework objects.
Use-Transaction takes a script block that can only contain commands
and expressions that use transaction-enabled .NET Framework objects,
such as instances of the
Microsoft.PowerShell.Commands.Management.TransactedString class.

The following command starts a transaction.

```
    start-transaction
```

The following New-Object command creates an instance of the
TransactedString class and saves it in the $t variable.

```
    $t = New-Object Microsoft.PowerShell.Commands.Management.TransactedString
```

The following command uses the Append method of the TransactedString
object to add text to the string. Because the command is not part
of the transaction, the change is effective immediately.

```
$t.append("Windows")
```

The following command uses the same Append method to add text, but
it adds the text as part of the transaction. The command is enclosed
in braces, and it is set as the value of the ScriptBlock parameter of
Use-Transaction. The UseTransaction parameter (UseTx) is required.

```
use-transaction {$t.append(" PowerShell")} -usetx
```

To see the current content of the transacted string in $t, use the
ToString method of the TransactedString object.

```
$t.tostring()
```

The output shows that only the non-transacted changes are effective.

Windows

To see the current content of the transacted string in $t from within the transaction, embed the expression in a Use-Transaction command.

use-transaction {$s.tostring()} -usetx

The output shows the transaction view.

Windows PowerShell

The following command commits the transaction.

complete-transaction

To see the final string:

$t.tostring()

Windows PowerShell

EXAMPLE 7: MANAGING MULTI-SUBSCRIBER TRANSACTIONS

When you start a transaction while another transaction is in progress, Windows PowerShell does not create a second transaction by default. Instead, it adds a subscriber to the current transaction.

This example shows how to view and manage a multi-subscriber transaction.

Begin by starting a transaction in the HKCU:\Software key.

start-transaction

The following command uses the Get-Transaction command to get the active transaction.

get-transaction

The result shows the object that represents the active transaction.

RollbackPreference SubscriberCount Status

```
------------------   ---------------   ------
Error            1             Active
```

The following command adds the MyCompany key to the registry. The command uses the UseTransaction parameter to include the command in the transaction.

```
new-item MyCompany -UseTransaction
```

The following command uses the Start-Transaction command to start a transaction. Although this command is typed at the command prompt, this scenario is more likely to happen when you run a script that contains a transaction.

```
start-transaction
```

A Get-Transaction command shows that the subscriber count on the transaction object is incremented. The value is now 2.

```
RollbackPreference   SubscriberCount   Status
------------------   ---------------   ------
Error            2             Active
```

The next command uses the New-ItemProperty cmdlet to add the MyKey registry entry to the MyCompany key. It uses the UseTransaction parameter to include the command in the transaction.

```
new-itemproperty -path MyCompany -name MyKey -UseTransaction
```

The MyCompany key does not exist in the registry, but this command succeeds because the two commands are part of the same transaction.

The following command commits the transaction. If it rolled back the transaction, the transaction would be rolled back for all the subscribers.

```
complete-transaction
```

A Get-Transaction command shows that the subscriber count on the transaction object is 1, but the value of Status is still Active (not Committed).

```
RollbackPreference   SubscriberCount   Status
------------------   ---------------   ------
Error                1                 Active
```

To finish committing the transaction, enter a second Complete-Transaction command. To commit a multi-subscriber transaction, you must enter one Complete-Transaction command for each Start-Transaction command.

```
complete-transaction
```

Another Get-Transaction command shows that the transaction has been committed.

```
RollbackPreference   SubscriberCount   Status
------------------   ---------------   ------
Error                0                 Committed
```

EXAMPLE 8: MANAGING INDEPENDENT TRANSACTIONS

When you start a transaction while another transaction is in progress, you can use the Independent parameter of Start-Transaction to make the new transaction independent of the original transaction.

When you do, Start-Transaction creates a new transaction object and makes the new transaction the active transaction.

Begin by starting a transaction in the HKCU:\Software key.

```
start-transaction
```

The following command uses the Get-Transaction command to get the active transaction.

```
get-transaction
```

The result shows the object that represents the active transaction.

```
RollbackPreference   SubscriberCount   Status
------------------   ---------------   ------
Error                1                 Active
```

The following command adds the MyCompany registry key as part of the transaction. It uses the UseTransaction parameter (UseTx) to include the command in the active transaction.

```
new-item MyCompany -use
```

The following command starts a new transaction. The command uses the Independent parameter to indicate that this transaction is not a subscriber to the active transaction.

```
start-transaction -independent
```

When you create an independent transaction, the new (most-recently created) transaction becomes the active transaction. You can use a Get-Transaction command to get the active transaction.

```
get-transaction
```

Note that the SubscriberCount of the transaction is 1, indicating that there are no other subscribers and that the transaction is new.

```
RollbackPreference   SubscriberCount   Status
------------------   ---------------   ------
Error                1                 Active
```

The new transaction must be finished (either committed or rolled back) before you can manage the original transaction.

The following command adds the MyOtherCompany key to the registry. It uses the UseTransaction parameter (UseTx) to include the command in the active transaction.

```
new-item MyOtherCompany -usetx
```

Now, roll back the transaction. If there were a single transaction with two subscribers, rolling back the transaction would roll back the entire transaction for all the subscribers.

However, because these transactions are independent, rolling back the newest transaction cancels the registry changes and makes the original transaction the active transaction.

```
undo-transaction
```

A Get-Transaction command confirms that the original transaction is still active in the session.

```
get-transaction

    RollbackPreference   SubscriberCount   Status
    ------------------   ---------------   ------
    Error                1                 Active
```

The following command commits the active transaction.

```
    complete-transaction
```

A Get-ChildItem command shows that the registry has been changed.

```
dir m*


    Hive: HKEY_CURRENT_USER\Software

    SKC  VC Name                           Property
    ---  -- ----                       --------
    83   1 Microsoft                    {(default)}
     0   0 MyCompany                      {}
```

SEE ALSO

- Start-Transaction
- Get-Transaction
- Complete-Transaction
- Undo-Transaction
- Use-Transaction
- Registry (provider)
- about_Providers
- Get-PSProvider
- Get-ChildItem

TOPIC

about_Trap

SHORT DESCRIPTION

Describes a keyword that handles a terminating error.

LONG DESCRIPTION

A terminating error stops a statement from running. If Windows PowerShell does not handle a terminating error in some way, Windows PowerShell also stops running the function or script in the current pipeline. In other languages, such as C#, terminating errors are referred to as exceptions.

The Trap keyword specifies a list of statements to run when a terminating error occurs. Trap statements handle the terminating errors and allow execution of the script or function to continue instead of stopping.

Syntax

The Trap statement has the following syntax:

```
trap [[<error type>]] {<statement list>}
```

The Trap statement includes a list of statements to run when a terminating error occurs. The Trap keyword can optionally specify an error type. An error type requires brackets.

A script or command can have multiple Trap statements. Trap statements can appear anywhere in the script or command.

Trapping All Terminating Errors

When a terminating error occurs that is not handled in another way in a script or command, Windows PowerShell checks for a Trap statement that handles the error. If a Trap statement is present, Windows PowerShell

continues running the script or command in the Trap statement.

The following example is a very simple Trap statement:

```
trap {"Error found."}
```

This Trap statement traps any terminating error. The following example is a function that contains this Trap statement:

```
function TrapTest {
   trap {"Error found."}
   nonsenseString
   }
```

This function includes a nonsense string that causes an error. Running this function returns the following:

```
C:\PS> TrapTest
Error found.
```

The following example includes a Trap statement that displays the error by using the $_ automatic variable:

```
function TrapTest {
   trap {"Error found: $_"}
   nonsenseString
   }
```

Running this version of the function returns the following:

```
C:\PS> TrapTest
Error found: The term 'nonsenseString' 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 then try again.
```

Trap statements can also be more complex. A Trap statement can include multiple conditions or function calls. It can log, test, or even run another program.

Trapping Specified Terminating Errors

The following example is a Trap statement that traps the CommandNotFoundException error type:

```
trap [System.Management.Automation.CommandNotFoundException]
  {"Command error trapped"}
```

When a function or script encounters a string that does not match a known command, this Trap statement displays the "Command error trapped" string. After running any statements in the Trap statement list, Windows PowerShell writes the error object to the error stream and then continues the script.

Windows PowerShell uses the Microsoft .NET Framework exception types. The following example specifies the System.Exception error type:

```
trap [System.Exception] {"An error trapped"}
```

The CommandNotFoundException error type inherits from the System.Exception type. This statement traps an error that is created by an unknown command. It also traps other error types.

You can have more than one Trap statement in a script. Each error can be trapped by only one Trap statement. If an error occurs, and more than one Trap statement is available, Windows PowerShell uses the Trap statement with the most specific error type that matches the error.

The following script example contains an error. The script includes a general Trap statement that traps any terminating error and a specific Trap statement that specifies the CommandNotFoundException type.

```
trap {"Other terminating error trapped" }
trap [System.Management.Automation.CommandNotFoundException] {"Command error
trapped"}
nonsenseString
```

Running this script produces the following result:

```
Command  error trapped
The term 'nonsenseString' 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 then try again.
At C:\PS>testScript1.ps1:3 char:19
+     nonsenseString <<<<
```

Because Windows PowerShell does not recognize "nonsenseString" as a cmdlet or other item, it returns a CommandNotFoundException error. This terminating error is trapped by the specific Trap statement.

The following script example contains the same Trap statements with a different error:

```
trap {"Other terminating error trapped" }
trap [System.Management.Automation.CommandNotFoundException]
   {"Command error trapped"}
1/$null
```

Running this script produces the following result:

```
Other terminating error trapped
Attempted to divide by zero.
At C:PS> errorX.ps1:3 char:7
+     1/ <<<< $null
```

The attempt to divide by zero does not create a CommandNotFoundException error. Instead, that error is trapped by the other Trap statement, which traps any terminating error.

Trapping Errors and Scope

If a terminating error occurs in the same scope as the Trap statement, after running the Trap statements, Windows PowerShell continues at the statement after the error. If the Trap statement is in a different scope from the error, execution continues at the next statement that is in the same scope as the Trap statement.

For instance, if an error occurs in a function, and the Trap statement is in the function, the script continues at the next statement. For example, the following script contains an error and a Trap statement:

```
function function1 {
   trap { "An error: " }
   NonsenseString
```

```
    "function1 was completed"
    }
```

Later in the script, running the Function1 function produces the following result:

```
function1
An error:
The term 'NonsenseString' 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
then try again.
At C:\PS>TestScript1.ps1:3 char:19
+    NonsenseString <<<<

function1 was completed
```

The Trap statement in the function traps the error. After displaying the message, Windows PowerShell resumes running the function. Note that Function1 was completed.

Compare this with the following example, which has the same error and Trap statement. In this example, the Trap statement occurs outside the function:

```
function function2 {
   NonsenseString
   "function2 was completed"
   }

trap { "An error: " }
   . . .
function2
```

Later in the script, running the Function2 function produces the following result:

```
An error:
The term 'NonsenseString' 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
then try again.
At C:\PS>TestScript2.ps1:4 char:19
+    NonsenseString <<<<
```

In this example, the "function2 was completed" command was not run. Although both terminating errors occur within a function, if the Trap statement is outside the function, Windows PowerShell does not go back into the function after the Trap statement runs.

Using the Break and Continue Keywords

You can use the Break and Continue keywords in a Trap statement to determine whether a script or command continues to run after a terminating error.

If you include a Break statement in a Trap statement list, Windows PowerShell stops the function or script. The following sample function uses the Break keyword in a Trap statement:

```
C:\PS> function break_example {
   trap {"Error trapped"; break;}
   1/$null
   "Function completed."
   }

C:\PS> break_example
Error trapped
Attempted to divide by zero.
At line:4 char:7
```

Because the Trap statement included the Break keyword, the function does not continue to run, and the "Function completed" line is not run.

If you include a Continue statement in a Trap statement, Windows PowerShell resumes after the statement that caused the error, just as it would without Break or Continue. With the Continue keyword, however, Windows PowerShell does not write an error to the error stream.

The following sample function uses the Continue keyword in a Trap statement:

```
C:\PS> function continue_example {
   trap {"Error trapped"; continue;}
   1/$null
   "Function completed."}
```

```
C:\PS> continue_example
Error trapped
Function completed.
```

The function resumes after the error is trapped, and the "Function completed" statement runs. No error is written to the error stream.

SEE ALSO

about_Break
about_Continue
about_Scopes
about_Throw
about_Try_Catch_Finally

TOPIC

about_Try_Catch_Finally

SHORT DESCRIPTION

Describes how to use the Try, Catch, and Finally blocks to handle terminating errors.

LONG DESCRIPTION

Use Try, Catch, and Finally blocks to respond to or handle terminating errors in scripts. The Trap statement can also be used to handle terminating errors in scripts. For more information, see about_Trap.

A terminating error stops a statement from running. If Windows PowerShell does not handle a terminating error in some way, Windows PowerShell also

stops running the function or script using the current pipeline. In other languages, such as C#, terminating errors are referred to as exceptions. For more information about errors, see about_Errors.

Use the Try block to define a section of a script in which you want Windows PowerShell to monitor for errors. When an error occurs within the Try block, the error is first saved to the $Error automatic variable. Windows PowerShell then searches for a Catch block to handle the error. If the Try statement does not have a matching Catch block, Windows PowerShell continues to search for an appropriate Catch block or Trap statement in the parent scopes. After a Catch block is completed or if no appropriate Catch block or Trap statement is found, the Finally block is run. If the error cannot be handled, the error is written to the error stream.

A Catch block can include commands for tracking the failure or for recovering the expected flow of the script. A Catch block can specify which error types it catches. A Try statement can include multiple Catch blocks for different kinds of errors.

A Finally block can be used to free any resources that are no longer needed by your script.

Try, Catch, and Finally resemble the Try, Catch, and Finally keywords used in the C# programming language.

Syntax

A Try statement contains a Try block, zero or more Catch blocks, and zero or one Finally block. A Try statement must have at least one Catch block or one Finally block.

The following shows the Try block syntax:

```
try {<statement list>}
```

The Try keyword is followed by a statement list in braces. If a terminating error occurs while the statements in the statement list are being run, the script passes the error object from the Try block to an appropriate Catch block.

The following shows the Catch block syntax:

```
catch [[<error type>][',' <error type>]*] {<statement list>}
```

Error types appear in brackets. The outermost brackets indicate the element is optional.

The Catch keyword is followed by an optional list of error type specifications and a statement list. If a terminating error occurs in the Try block, Windows PowerShell searches for an appropriate Catch block. If one is found, the statements in the Catch block are executed.

The Catch block can specify one or more error types. An error type is a Microsoft .NET Framework exception or an exception that is derived from a .NET Framework exception. A Catch block handles errors of the specified .NET Framework exception class or of any class that derives from the specified class.

If a Catch block specifies an error type, that Catch block handles that type of error. If a Catch block does not specify an error type, that Catch block handles any error encountered in the Try block. A Try statement can include multiple Catch blocks for the different specified error types.

The following shows the Finally block syntax:

```
finally {<statement list>}
```

The Finally keyword is followed by a statement list that runs every time the script is run, even if the Try statement ran without error or an error was caught in a Catch statement.

Note that pressing CTRL+C stops the pipeline. Objects that are sent to the pipeline will not be displayed as output. Therefore, if you include a statement to be displayed, such as "Finally block has run", it will not be displayed after you press CTRL+C, even if the Finally block ran.

Catching Errors

The following sample script shows a Try block with a Catch block:

```
try { NonsenseString }
```

```
catch { "An error occurred." }
```

The Catch keyword must immediately follow the Try block or another Catch block.

Windows PowerShell does not recognize "NonsenseString" as a cmdlet or other item. Running this script returns the following result:

```
An error occurred.
```

When the script encounters "NonsenseString", it causes a terminating error. The Catch block handles the error by running the statement list inside the block.

Using Multiple Catch Statements

A Try statement can have any number of Catch blocks. For example, the following script has a Try block that downloads MyFile.doc, and it contains two Catch blocks:

```
try
{
  $wc = new-object System.Net.WebClient
  $wc.DownloadFile("http://www.contoso.com/MyDoc.doc")
}
catch [System.Net.WebException],[System.IO.IOException]
{
   "Unable to download MyDoc.doc from http://www.contoso.com."
}
catch
{
   "An error occurred that could not be resolved."
}
```

The first Catch block handles errors of the System.Net.WebException and System.IO.IOException types. The second Catch block does not specify an error type. The second Catch block handles any other terminating errors that occur.

Windows PowerShell matches error types by inheritance. A Catch block handles errors of the specified .NET Framework exception class or of any class that derives from the specified class. The following example contains a Catch block that catches a "Command Not Found" error:

```
    catch [System.Management.Automation.CommandNotFoundException]
      {"Inherited Exception" }
```

The specified error type, CommandNotFoundException, inherits from the System.SystemException type. The following example also catches a Command Not Found error:

```
    catch [System.SystemException] {"Base Exception" }
```

This Catch block handles the "Command Not Found" error and other errors that inherit from the SystemException type.

If you specify an error class and one of its derived classes, place the Catch block for the derived class before the Catch block for the general class.

Freeing Resources by Using Finally

To free resources used by a script, add a Finally block after the Try and Catch blocks. The Finally block statements run regardless of whether the Try block encounters a terminating error. Windows PowerShell runs the Finally block before the script terminates or before the current block goes out of scope.

A Finally block runs even if you use CTRL+C to stop the script. A Finally block also runs if an Exit keyword stops the script from within a Catch block.

SEE ALSO

about_Break
about_Continue
about_Scopes
about_Throw
about_Trap

TOPIC

about_Types.ps1xml

SHORT DESCRIPTION

Explains how to use Types.ps1xml files to extend the types of objects that are used in Windows PowerShell.

LONG DESCRIPTION

Extended type data defines additional properties and methods ("members") of object types in Windows PowerShell. There are two techniques for adding extended type data to a Windows PowerShell session.

-- Types.ps1xml file: An XML file that defines extended type data.
-- Update-TypeData: A cmdlet that reloads Types.ps1xml files and defines extended data for types in the current session.

This topic describes Types.ps1xml files. For more information about using the Update-TypeData cmdlet to add dynamic extended type data to the current session see Update-TypeData (http://go.microsoft.com/fwlink/?LinkID=113421).

About Extended Type Data

Extended type data defines additional properties and methods ("members") of object types in Windows PowerShell. You can extend any type that is supported by Windows PowerShell and use the added properties and methods in the same way that you use the properties that are defined on the object types.

For example, Windows PowerShell adds a DateTime property to all System.DateTime objects, such as the ones that the Get-Date cmdlet returns.

```
PS C:\> (Get-Date).DateTime
Sunday, January 29, 2012 9:43:57 AM
```

You won't find the DateTime property in the description of the System.DateTime structure (http://msdn.microsoft.com/library/system.datetime.aspx), because Windows PowerShell adds the property and it is visible only in Windows PowerShell.

To add the DateTime property to all Windows PowerShell sessions, Windows PowerShell defines the DateTime property in the Types.ps1xml file in the Windows PowerShell installation directory ($pshome).

Adding Extended Type Data to Windows PowerShell.

There are three sources of extended type data in Windows PowerShell sessions.

-- The Types.ps1xml files in the Windows PowerShell installation directory are loaded automatically into every Windows PowerShell session.

-- The Types.ps1xml files that modules export are loaded when the module is imported into the current session.

-- Extended type data that is defined by using the Update-TypeData cmdlet is added only to the current session. It is not saved in a file.

In the session, the extended type data from the three sources is applied to objects in the same way and is available on all objects of the specified types.

The TypeData Cmdlets

The following TypeData cmdlets are included in the Microsoft.PowerShell.Utility module in Windows PowerShell 3.0 and later versions of Windows PowerShell.

Get-TypeData: Gets extended type data in the current session.
Update-TypeData: Reloads Types.ps1xml files. Adds extended type data to the current session.
Remove-TypeData: Removes extended type data from the current session.

For more information about these cmdlets, see the help topic for each cmdlet.

Built-in Types.ps1xml Files

The Types.ps1xml files in the $pshome directory are added automatically to every session.

The Types.ps1xml file in the Windows PowerShell installation directory ($pshome) is an XML-based text file that lets you add properties and methods to the objects that are used in Windows PowerShell. Windows PowerShell has built-in Types.ps1xml files that add several elements to the .NET Framework types, but you can create additional Types.ps1xml files to further extend the types.

For example, by default, array objects (System.Array) 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 an alias property named "Count" that displays the same value. The following XML adds the Count property to the System.Array type.

```
<Type>
  <Name>System.Array</Name>
  <Members>
    <AliasProperty>
      <Name>Count</Name>
      <ReferencedMemberName>
        Length
      </ReferencedMemberName>
    </AliasProperty>
  </Members>
</Type>
```

To get the new AliasProperty, use a Get-Member command on any array, as shown in the following example.

```
Get-Member -inputobject (1,2,3,4)
```

The command returns the following results.

```
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 )
...
```

As a result, you can use either the Count property or the Length property of arrays in Windows PowerShell. For example:

```
C:\PS> (1, 2, 3, 4).count
4

C:\PS> (1, 2, 3, 4).length
4
```

Creating New Types.ps1xml Files

The .ps1xml files that are installed with Windows PowerShell are digitally signed to prevent tampering because the formatting can include script blocks. Therefore, to add a property or method to a .NET Framework type, create your own Types.ps1xml files, and then add them to your Windows PowerShell session.

To create a new file, start by copying an existing Types.ps1xml file. The new file can have any name, but it must have a .ps1xml file name extension. 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.

When you have saved the new file, use the Update-TypeData cmdlet to add the new file to your Windows PowerShell session. If you want your types to take precedence over the types that are defined in the built-in file, use the PrependData parameter of the Update-TypeData cmdlet. Update-TypeData affects only the current session. To make the change to all future sessions, export the console, or add the Update-TypeData command to your Windows PowerShell profile.

Types.ps1xml and Add-Member

The Types.ps1xml files add properties and methods to all the instances of the objects of the specified .NET Framework type in the affected Windows PowerShell session. However, if you need to add properties or methods only to one instance of an object, use the Add-Member cmdlet.

For more information,see Add-Member.

Example: Adding an Age Member to FileInfo Objects

This example shows how to add an Age property to file objects (System.IO.FileInfo). The age of a file is the difference between its creation time and the current time in days.

It is easiest to use the original Types.ps1xml file as a template for the new file. The following command copies the original file to a file called MyTypes.ps1xml in the $pshome directory.

```
copy-item Types.ps1xml MyTypes.ps1xml
```

Next, open the Types.ps1xml file in any XML or text editor, such

as Notepad. Because the Age property is calculated by using a script block, find a <ScriptProperty> tag to use as a model for the new Age property.

Copy the XML between the <Type> and </Type> tags of the code to create the script property. Then, delete the remainder of the file, except for the opening <?xml> and <Types> tags and the closing </Types> tag. You must also delete the digital signature to prevent errors.

Begin with the model script property, such as the following script property, which was copied from the original Types.ps1xml file.

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
  <Type>
    <Name>System.Guid</Name>
      <Members>
        <ScriptProperty>
          <Name>Guid</Name>
          <GetScriptBlock>$this.ToString()</GetScriptBlock>
        </ScriptProperty>
      </Members>
  </Type>
</Types>
```

Then, change the name of the .NET Framework type, the name of the property, and the value of the script block to create an Age property for file objects.

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
  <Type>
    <Name>System.IO.FileInfo</Name>
      <Members>
        <ScriptProperty>
          <Name>Age</Name>
          <GetScriptBlock>
            ((get-date) - ($this.creationtime)).days
          </GetScriptBlock>
        </ScriptProperty>
      </Members>
  </Type>
</Types>
```

After you save the file and close it, run an Update-TypeData command,

such as the following command, to add the new Types.ps1xml file to the current session. The command uses the PrependData parameter to place the new file in a higher precedence order than the original file. (For more information about Update-TypeData, see Update-TypeData.)

```
update-typedata -prependpath $pshome\MyTypes.ps1xml
```

To test the change, run a Get-ChildItem command to get the PowerShell.exe file in the $pshome directory, and then pipe the file to the Format-List cmdlet to list all of the properties of the file. As a result of the change, the Age property appears in the list.

```
get-childitem $pshome\PowerShell.exe | format-list -property *


PSPath          : Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS...
PSParentPath    : Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS...
PSChildName     : PowerShell.exe
PSDrive         : C
PSProvider      : Microsoft.PowerShell.Core\FileSystem
PSIsContainer   : False
Age             : 16
VersionInfo     : File:           C:\WINDOWS\system32\WindowsPow...
                  InternalName:   POWERSHELL
                  OriginalFilename: PowerShell.EXE
...
```

You can also display the Age property of the file by using the following command.

```
(get-childitem $pshome\PowerShell.exe).age
    16
```

The XML in Types.ps1xml Files

The <Types> tag encloses all of the types that are defined in the file. There should be only one pair of <Types> tags.

Each .NET Framework type mentioned in the file should be represented by a pair of <Type> tags.

The type tags must contain the following tags:

<Name>: A pair of <Name> tags that enclose the name of the affected .NET Framework type.

<Members>: A pair of <Members> tags that enclose the tags for the new properties and methods that are defined for the .NET Framework type.

Any of the following member tags can be inside the <Members> tags.

<AliasProperty>: Defines a new name for an existing property.

The <AliasProperty> tag must have a pair of <Name> tags that specify the name of the new property and a pair of <ReferencedMemberName> tags that specify the existing property.

For example, the Count alias property is an alias for the Length property of array objects.

```
<Type>
  <Name>System.Array</Name>
  <Members>
    <AliasProperty>
      <Name>Count</Name>
      <ReferencedMemberName>Length</ReferencedMemberName>
    </AliasProperty>
  </Members>
</Type>
```

<CodeMethod>: References a static method of a .NET Framework class.

The <CodeMethod> tag must have a pair of <Name> tags that specify the name of the new method and a pair of <GetCodeReference> tags that specify the code in which the method is defined.

For example, the Mode property of directories (System.IO.DirectoryInfo objects) is a code property defined in the Windows PowerShell FileSystem provider.

```
<Type>
  <Name>System.IO.DirectoryInfo</Name>
  <Members>
    <CodeProperty>
      <Name>Mode</Name>
      <GetCodeReference>
        <TypeName>Microsoft.PowerShell.Commands.FileSystemProvider</TypeName>
        <MethodName>Mode</MethodName>
      </GetCodeReference>
    </CodeProperty>
  </Members>
</Type>
```

<CodeProperty>: References a static method of a .NET Framework class.

The <CodeProperty> tag must have a pair of <Name> tags that specify the name of the new property and a pair of <GetCodeReference> tags that specify the code in which the property is defined.

For example, the Mode property of directories (System.IO.DirectoryInfo objects) is a code property defined in the Windows PowerShell FileSystem provider.

```
<Type>
  <Name>System.IO.DirectoryInfo</Name>
  <Members>
    <CodeProperty>
      <Name>Mode</Name>
      <GetCodeReference>
        <TypeName>Microsoft.PowerShell.Commands.FileSystemProvider</TypeName>
        <MethodName>Mode</MethodName>
      </GetCodeReference>
    </CodeProperty>
  </Members>
</Type>
```

<MemberSet>: Defines a collection of members (properties and methods).

The <MemberSet> tags appear within the primary <Members> tags. The tags must enclose a pair of <Name> tags surrounding the name of the member set and a pair of secondary <Members> tags that surround the members (properties and methods) in the set. Any of the tags that create a property (such as <NoteProperty> or <ScriptProperty>) or a method (such as <Method> or <ScriptMethod>) can be members of the set.

In Types.ps1xml files, the <MemberSet> tag is used to define the default views of the .NET Framework objects in Windows PowerShell. In this case, the name of the member set (the value within the <Name> tags) is always "PsStandardMembers", and the names of the properties (the value of the <Name> tag) are one of the following:

- DefaultDisplayProperty: A single property of an object.

- DefaultDisplayPropertySet: One or more properties of an object.

- DefaultKeyPropertySet: One or more key properties of an object. A key property identifies instances of property values, such as the ID number of items in a session history.

For example, the following XML defines the default display of services (System.ServiceProcess.ServiceController objects) that are returned by the Get-Service cmdlet. It defines a member set named "PsStandardMembers" that consists of a default property set with the Status, Name, and DisplayName properties.

```
<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>
```

<Method>: References a native method of the underlying object.

<Methods>: A collection of the methods of the object.

<NoteProperty>: Defines a property with a static value.

The <NoteProperty> tag must have a pair of <Name> tags that specify the name of the new property and a pair of <Value> tags that specify the value of the property.

For example, the following XML creates a Status property for directories (System.IO.DirectoryInfo objects). The value of the Status property is always "Success".

```
<Type>
  <Name>System.IO.DirectoryInfo</Name>
  <Members>
    <NoteProperty>
      <Name>Status</Name>
  <Value>Success</Value>
    </NoteProperty>
```

```
    </Members>
  </Type>
```

<ParameterizedProperty>: Properties that take arguments and return a value.

<Properties>: A collection of the properties of the object.

<Property>: A property of the base object.

<PropertySet>: Defines a collection of properties of the object.

The <PropertySet> tag must have a pair of <Name> tags that specify the name of the property set and a pair of <ReferencedProperty> tags that specify the properties. The names of the properties are enclosed in <Name> tag pairs.

In Types.ps1xml, <PropertySet> tags are used to define sets of properties for the default display of an object. You can identify the default displays by the value "PsStandardMembers" in the <Name> tag of a <MemberSet> tag.

For example, the following XML creates a Status property for directories (System.IO.DirectoryInfo objects). The value of the Status property is always "Success".

```
  <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>
```

<ScriptMethod>: Defines a method whose value is the output of a script.

The <ScriptMethod> tag must have a pair of <Name> tags that specify the name of the new method and a pair of <Script> tags that enclose the script block that returns the method result.

For example, the ConvertToDateTime and ConvertFromDateTime methods of management objects (System.System.Management.ManagementObject) are script methods that use the ToDateTime and ToDmtfDateTime static methods of the System.Management.ManagementDateTimeConverter class.

```
<Type>
  <Name>System.Management.ManagementObject</Name>
  <Members>
    <ScriptMethod>
      <Name>ConvertToDateTime</Name>
      <Script>
        [System.Management.ManagementDateTimeConverter]::ToDateTime($args[0])
      </Script>
    </ScriptMethod>
    <ScriptMethod>
      <Name>ConvertFromDateTime</Name>
      <Script>
        [System.Management.ManagementDateTimeConverter]::ToDmtfDateTime($args[0])
      </Script>
    </ScriptMethod>
  </Members>
</Type>
```

<ScriptProperty>: Defines a property whose value is the output of a script.

The <ScriptProperty> tag must have a pair of <Name> tags that specify the name of the new property and a pair of <GetScriptBlock> tags that enclose the script block that returns the property value.

For example, the VersionInfo property of files (System.IO.FileInfo objects) is a script property that results from using the FullName property of the GetVersionInfo static method of System.Diagnostics.FileVersionInfo objects.

```
<Type>
  <Name>System.IO.FileInfo</Name>
  <Members>
    <ScriptProperty>
      <Name>VersionInfo</Name>
      <GetScriptBlock>
        [System.Diagnostics.FileVersionInfo]::GetVersionInfo($this.FullName)
```

```
            </GetScriptBlock>
          </ScriptProperty>
        </Members>
      </Type>
```

For more information, see the Windows PowerShell Software Development Kit (SDK) in the MSDN (Microsoft Developer Network)library at http://go.microsoft.com/fwlink/?LinkId=144538.

Update-TypeData

To load your Types.ps1xml files into a Windows PowerShell session, run the Update-TypeData cmdlet. If you want the types in your file to take precedence over types in the built-in Types.ps1xml file, add the PrependData parameter of Update-TypeData. Update-TypeData affects only the current session. To make the change to all future sessions, export the session, or add the Update-TypeData command to your Windows PowerShell profile.

Exceptions that occur in properties, or from adding properties to an Update-TypeData command, do not report errors to StdErr. This is to suppress exceptions that would occur in many common types during formatting and outputting. If you are getting .NET Framework properties, you can work around the suppression of exceptions by using method syntax instead, as shown in the following example:

```
"hello".get_Length()
```

Note that method syntax can only be used with .NET Framework properties. Properties that are added by running the Update-TypeData cmdlet cannot use method syntax.

Signing a Types.ps1xml File

To protect users of your Types.ps1xml file, you can sign the file using a digital signature. For more information, see about_Signing.

SEE ALSO

about_Signing (http://go.microsoft.com/fwlink/?LinkID=113268)
Copy-Item (http://go.microsoft.com/fwlink/?LinkID=113292)
Copy-ItemProperty (http://go.microsoft.com/fwlink/?LinkID=113293)
Get-Member (http://go.microsoft.com/fwlink/?LinkID=113322)
Get-TypeData (http://go.microsoft.com/fwlink/?LinkID=217033)
Remove-TypeData (http://go.microsoft.com/fwlink/?LinkID=217038)
Update-TypeData (http://go.microsoft.com/fwlink/?LinkID=113421)

TOPIC

about_Type_Operators

SHORT DESCRIPTION

Describes the operators that work with Microsoft .NET Framework types.

LONG DESCRIPTION

The Boolean type operators (-is and -isNot) tell whether an object is an instance of a specified .NET Framework type. The -is operator returns a value of TRUE if the type matches and a value of FALSE otherwise. The -isNot operator returns a value of FALSE if the type matches and a value of TRUE otherwise.

The -as operator tries to convert the input object to the specified .NET Framework type. If it succeeds, it returns the converted object. It if fails, it returns nothing. It does not return an error.

The following table lists the type operators in Windows PowerShell.

Operator	Description	Example
-is	Returns TRUE when the input is an instance of the specified .NET Framework type.	C:\PS> (get-date) -is [DateTime] True
-isNot	Returns TRUE when the input is not an instance of the specified .NET Framework type.	C:\PS> (get-date) -isNot [DateTime] False

-as	Converts the input to the specified .NET Framework type.	C:\PS> 12/31/07 -as [DateTime] Monday, December 31, 2007 12:00:00 AM

The syntax of the type operators is as follows:

```
<input> <operator> [.NET type]
```

You can also use the following syntax:

```
<input> <operator> ".NET type"
```

To specify the .NET Framework type, enclose the type name in brackets ([ ]), or enter the type as a string, such as [DateTime] or "DateTime" for System.DateTime. If the type is not at the root of the system namespace, specify the full name of the object type. You can omit "System.". For example, to specify System.Diagnostics.Process, enter [System.Diagnostics.Process], [Diagnostics.Process], or "diagnostics.process".

The type operators always return a Boolean value, even if the input is a collection of objects. However, when the input is a collection, the type operators match the .NET Framework type of the collection. They do not match the type of each object, even when all of the objects are of the same type.

To find the .NET Framework type of an object, use the Get-Member cmdlet. Or, use the GetType method of all the objects together with the FullName property of this method. For example, the following statement gets the type of the return value of a Get-Culture command:

```
C:\PS> (get-culture).gettype().fullname
System.Globalization.CultureInfo
```

EXAMPLES

The following examples show some uses of the Type operators:

```
C:\PS> 32 -is [Float]
False

C:\PS> 32 -is "int"
True

C:\PS> (get-date) -is [DateTime]
```

```
True

C:\PS> "12/31/2007" -is [DateTime]
False

C:\PS> "12/31/2007" -is [String]
True

C:\PS> (get-process PowerShell)[0] -is [System.Diagnostics.Process]
True

C:\PS> (get-command get-member) -is [System.Management.Automation.CmdletInfo]
True
```

The following example shows that when the input is a collection of objects, the matching type is the .NET Framework type of the collection, not the type of the individual objects in the collection.

In this example, although both the Get-Culture and Get-UICulture cmdlets return System.Globalization.CultureInfo objects, a collection of these objects is a System.Object array.

```
C:\PS> (get-culture) -is [System.Globalization.CultureInfo]
True

C:\PS> (get-uiculture) -is [System.Globalization.CultureInfo]
True

C:\PS> (get-culture), (get-uiculture) -is [System.Globalization.CultureInfo]
False

C:\PS> (get-culture), (get-uiculture) -is [Array]
True

C:\PS> (get-culture), (get-uiculture) | foreach {$_ -is [System.Globalization.CultureInfo]}
True
True

C:\PS> (get-culture), (get-uiculture) -is [Object]
True
```

The following examples show how to use the -as operator.

```
C:\PS> "12/31/07" -is [DateTime]
False
```

```
C:\PS> "12/31/07" -as [DateTime]
Monday, December 31, 2007 12:00:00 AM

C:\PS> $date = "12/31/07" -as [DateTime]

C:\PS>$a -is [DateTime]
True

C:\PS> 1031 -as [System.Globalization.CultureInfo]

LCID          Name          DisplayName
----          ----          -----------
1031          de-DE         German (Germany)
```

The following example shows that when the -as operator cannot convert the input object to the .NET Framework type, it returns nothing.

```
C:\PS> 1031 -as [System.Diagnostic.Process]
C:\PS>
```

SEE ALSO

about_Operators

TOPIC

About_Updatable_Help

SHORT DESCRIPTION

Describes the updatable help system in Windows PowerShell.

LONG DESCRIPTION

Windows PowerShell provides several different ways to access the most up-to-date help topics for Windows PowerShell cmdlets

and concepts.

The Updatable Help system, introduced in Windows PowerShell 3.0, is designed to assure that you always have the newest help topics on your local computer so that you can read them at the command line. It makes it easy to download and install help files and to update them whenever newer help files become available.

To provide updated help for multiple computers in an enterprise and for computers that do not have access to the Internet, Updatable Help lets you download help files to a file system directory or file share, and then install the help files from the file share.

In Windows PowerShell 4.0, the HelpInfoUri property is preserved over Windows PowerShell remoting, which allows Save-Help to work for modules that are installed on a remote computer, but are not necessarily installed on the local computer. You can save a PSModuleInfo object to disk or removable media (such as a USB drive) by running Export-CliXml on a computer that does not have Internet access, importing the PSModuleInfo object on a computer that does have Internet access, and then running Save-Help on the PSModuleInfo object. The saved help can be copied to the remote, disconnected computer by using removable media, and then installed by running Update-Help. These improvements in Save-Help functionality let you install help on computers that are without any kind of network access. For an example of how to use the new Save-Help functionality, see "HOW TO UPDATE HELP FROM A FILE SHARE: SAVE-HELP" in this topic.

Updatable Help also supports online access to the newest help topics and basic help for cmdlets, even when there are no help files on the computer.

Windows PowerShell 3.0 does not come with Help files. You can use the Updatable Help feature to install the help files for all of the commands that are included by default in Windows PowerShell and for all Windows modules.

UPDATABLE HELP CMDLETS

Update-Help: Downloads the newest help files from the Internet or a file share, and installs them on the local computer.

Save-Help: Downloads the newest help files from the Internet and saves them in a file system directory or file share. To install the help files on computers, use Update-Help.

Get-Help: -- Displays help topics at the command line.
-- Gets help from the help files on the computer.
-- Displays auto-generated help for cmdlets and functions that do not have help files.
-- Opens online help topics for cmdlets, functions, scripts, and workflows in your default Internet browser.

UPDATE HELP IN WINDOWS POWERSHELL ISE

You can also update help by using the "Update Windows PowerShell Help" item in the Help menu in Windows PowerShell Integrated Scripting Environment (ISE).

The "Update Windows PowerShell Help" item runs an Update-Help command without parameters.

AUTO-GENERATED HELP: HELP WITHOUT HELP FILES

If you do not have the help file for a cmdlet, function, or workflow on the computer, the Get-Help cmdlet displays auto-generated help and prompts you to download the help files or read them online.

Auto-generated help includes syntax and aliases, and remarks that explain how to use the Updatable Help cmdlets and to access the online help topics.

For example, the following command gets basic help for the Get-Culture cmdlet. The output shows the Get-Help display when there are no help files on the computer.

```
PS C:\> Get-Help Get-Culture

NAME
  Get-Culture

SYNTAX
  Get-Culture [<CommonParameters>]

ALIASES
  None

REMARKS
  To get the latest Help content including descriptions
  and examples type: Update-Help.
```

HELP FILES FOR MODULES

The smallest unit of Updatable Help is help for a module. Module help includes help for all of the cmdlets, functions, workflows, providers, scripts, and concepts in a module. You can update help for all modules that are installed on the computer, even if they are not imported into the current session.

You can update help for the entire module, but you cannot update help for individual cmdlets.

To find the module that contains a particular cmdlet, use the following command format:

(Get-Command <cmdlet-name>).ModuleName

For example, to find the module that contains the Set-ExecutionPolicy cmdlet, type:

(Get-Command Set-ExecutionPolicy).ModuleName

To update help for a particular module, type:

Update-Help -Module <ModuleName>

For example, to update help for the module that contains the Set-ExecutionPolicy cmdlet, type:

Update-Help -Module Microsoft.PowerShell.Security

PERMISSIONS FOR UPDATABLE HELP

To update help for the modules in the $pshome\Modules directory, you must be member of the Administrators group on the computer.

If you are not a member of the Administrators group, you cannot update help for these modules; but if you have Internet access, you can view help online in the TechNet Library.

Updating help for modules in the $home\Documents\WindowsPowerShell\Modules directory or modules in other subdirectories of the $home

directory do not require special permissions.

The Update-Help and Save-Help cmdlets have a UseDefaultCredentials parameter that provides the explicit credentials of the current user. This parameter is designed for accessing secure Internet locations.

The Update-Help and Save-Help cmdlets also have a Credential parameter that allows you to run the command on a remote computer and access a file share on a third computer. The Credential parameter is valid only when you use the SourcePath or LiteralPath parameters of Update-Help and the DestinationPath or LiteralPath parameters of Save-Help.

HOW TO INSTALL AND UPDATE HELP FILES

To download and install help files for the first time, or to update the help files on your computer, use the Update-Help cmdlet.

The Update-Help cmdlet does all of the hard work for you, including the following tasks.

-- Determines which modules support Updatable Help.
-- Finds the Internet location where each module stores its Updatable Help files.
-- Compares the help files for each module on your computer to the newest help files that are available for each module.
-- Downloads the new files from the Internet.
-- Unwraps the help file package.
-- Verifies that the files are valid help files.
-- Installs the help files in the language-specific subdirectory of the module directory.

To access the new help topics, use the Get-Help cmdlet. You do not need to restart Windows PowerShell.

To install or update help for all modules on the computer that supports Updatable Help, type:

```
Update-Help
```

To update help for particular modules, add the Module parameter of Update-Help. Wildcard characters are permitted in the module name.

For example, to update help for the ServerManager module, type:

```
Update-Help -Module ServerManager
```

NOTES
Without parameters, Update-Help updates help for all modules in the session and for all installed modules that support Updatable Help. To be included, modules must be installed in directories that are listed in the value of the PSModulePath environment variable. These are also modules that are returned by a "Get-Help -ListAvailable" command.

If the value of the Module parameter is * (all), Update-Help attempts to update help for all installed modules, including modules that do not support Updatable Help. This command typically generates many errors as the cmdlet encounters modules that do not support Updatable Help.

HOW TO UPDATE HELP FROM A FILE SHARE: SAVE-HELP

To support computers that are not connected to the Internet, or to control or streamline help updating in an enterprise, use the Save-Help cmdlet. The Save-Help cmdlet downloads help files from the Internet and saves them in a file system directory that you specify.

Save-Help compares the help files in the specified directory to the newest help files that are available for each module. If the directory has no help files or newer help files are available for the module, the Save-Help cmdlet downloads the new files from the Internet. However, it does not unwrap or install the help files.

To install or update the help files on a computer from help files that were saved to a file system directory, use the SourcePath parameter of the Update-Help cmdlet. The Update-Help cmdlet identifies the newest help files, unwraps and validates them, and installs them in the language-specific subdirectories of the module directories.

For example, to save help for all installed modules to the \\Server\Share directory, type:

```
Save-Help -DestinationPath \\Server\Share
```

Then, to update help from the \\Server\Share directory, type:

```
Update-Help -SourcePath \\Server\Share
```

The following examples show the use of Save-Help to save help

for modules that are not installed on the local computer. In
this example, the administrator runs Save-Help to save the help
for the DhcpServer module from an Internet-connected client computer,
without installing the DhcpServer module or DHCP Server role on the
local computer.

```
    # Option 1: Run Invoke-Command to get the PSModuleInfo object for
      the remote module, save it in a variable, $m, and then run Save-Help
      on the PSModuleInfo object by specifying the variable $m as the
      module name.

    $m = Invoke-Command -ComputerName RemoteServer -ScriptBlock
    { Get-Module -Name DhcpServer -ListAvailable }
    Save-Help -Module $m -DestinationPath C:\SavedHelp


    # Option 2: Open a PSSession, targeted at the computer that is
      running the DHCP Server module, to get the PSModuleInfo object for
      the module, save it in a variable $m, and then run Save-Help on the
      object that is saved in the $m variable.

    $s = New-PSSession -ComputerName RemoteServer
    $m = Get-Module -PSSession $s -Name DhcpServer -ListAvailable
    Save-Help -Module $m -DestinationPath C:\SavedHelp


    # Option 3: Open a CIM session, targeted at the computer that is
      running the DHCP Server module, to get the PSModuleInfo object for
      the module, save it in a variable $m, and then run Save-Help on the
      object that is saved in the $m variable.

    $c = New-CimSession -ComputerName RemoteServer
    $m = Get-Module -CimSession $c -Name DhcpServer -ListAvailable
    Save-Help -Module $m -DestinationPath C:\SavedHelp
```

In the following example, the administrator installs help for the
DHCP Server module on a computer that does not have network access.

```
    # First, run Export-CliXml to export the PSModuleInfo object to
      a shared folder or to removable media.

    $m = Get-Module -Name DhcpServer –ListAvailable
    Export-CliXml –Path E:\UsbFlashDrive\DhcpModule.xml –InputObject $m

    # Next, transport the removable media to a computer that has
      Internet access, and then import the PSModuleInfo object with
      Import-CliXml. Run Save-Help to save the Help for the imported
      DhcpServer module PSModuleInfo object.
```

```
$deserialized_m = Import-CliXml E:\UsbFlashDrive\DhcpModule.xml
Save-Help -Module $deserialized_m –DestinationPath
  E:\UsbFlashDrive\SavedHelp

# Finally, transport the removable media back to the computer that
  does not have network access, and then install the help by running
  Update-Help.

Update-Help –Module DhcpServer –SourcePath
  E:\UsbFlashDrive\SavedHelp
```

NOTES:

Without parameters, Save-Help downloads help for all modules in the session and for all installed modules that support Updatable Help. To be included, modules must be installed in directories that are listed in the value of the PSModulePath environment variable, on either the local computer or on a remote computer for which you want to save help. These are also modules that are returned by running a "Get-Help -ListAvailable" command.

HOW TO UPDATE HELP FILES IN DIFFERENT LANGUAGES

By default, the Update-Help and Save-Help cmdlets download help in the UI culture and language that is set for Windows on the local computer. If help files for the specified modules are not available in the local UI culture, Update-Help and Save-Help use the Windows language fallback rules to find the best supported language.

However, you can use the UICulture parameters of the Update-Help and Save-Help cmdlets to download and install help files in any UI cultures in which they are available.

For example, to save the newest help files for all modules on the session in Japanese (Ja-jp) and French (fr-FR), type:

```
Save-Help -Path \\Server\Share -UICulture ja-jp, fr-fr
```

If help files for the modules are not available in the languages that you specified, the Update-Help and Save-Help cmdlets return an error message that lists the languages in which help for each module is available so you can choose the alternative that best meets your needs.

HOW TO UPDATE HELP AUTOMATICALLY

To assure that you always have the newest help files, you can
add an Update-Help command to your Windows PowerShell profile.

An internal quota prevents the Update-Help command from running
more than once each day. To override the once-per-day maximum,
use the Force parameter.

Use a command like the following one in your profile. This command
updates help for all installed modules in a background job so it
doesn't disturb your work. It uses an Out-Null command to suppress
the job that is returned and any error messages that would otherwise
appear when you use the command more than once per day.

```
Start-Job {Update-Help} | Out-Null
```

You can also create a scheduled job that runs the Update-Help or
Save-Help cmdlet at any interval.

For example, the following command creates a scheduled job that runs an
Update-Help help command every Friday at 5:00 AM. To run this command,
start Windows PowerShell with the "Run as administrator" option.

```
Register-ScheduledJob -Name UpdateHelpJob -ScriptBlock {Update-Help} `
  -Trigger (New-JobTrigger -Weekly -DaysOfWeek Friday -At "5:00 AM")
```

For more information about scheduled jobs, see about_Scheduled_Jobs.

HOW TO USE ONLINE HELP

If you cannot or choose not to update the help files on your
local computer, you can still get the newest help files online.

To open the online help topic for any cmdlet or function, use
the Online parameter of the Get-Help cmdlet.

For example, the following command opens the online help topic
for the Get-Job cmdlet in your default Internet browser:

```
Get-Help Get-Job -Online
```
-or-
```
Get-Help -on Get-Job
```

To get online help for a script, use the Online parameter and
the full path to the script.

The Online parameter does not work with About topics. To see the about topics for Windows PowerShell Core, including help topics about the Windows PowerShell language, see "Windows PowerShell Core Module About Topics" at http://go.microsoft.com/fwlink/?LinkID=113206.

HOW TO MINIMIZE OR PREVENT INTERNET DOWNLOADS

To minimize Internet downloads and provide Updatable Help to users who are not connected to the Internet, use the Save-Help cmdlet. Download help from the Internet and save it to a network share. Then, create a Group Policy setting or scheduled job that runs an Update-Help command on all computers. Set the value of the SourcePath parameter of the Update-Help cmdlet to the network share.

To prevent users who have Internet access from downloading Updatable Help from the Internet, use the "Set the default source path for Update-Help" Group Policy setting.

This Group Policy setting implicitly adds the SourcePath parameter, with the file system location that you specify, to every Update-Help command on every affected computer. Users can use the SourcePath parameter explicitly to specify a different file system location, but they cannot exclude the SourcePath parameter and download help from the Internet.

NOTE: The "Set the default source path for Update-Help" group policy setting appears under Computer Configuration and User Configuration. However, only the policy setting under Computer Configuration is effective. The policy setting under User Configuration is ignored.

For more information, see about_Group_Policy_Settings.

HOW TO UPDATE HELP FOR NON-STANDARD MODULES

To update or save help for a module that is not returned by the ListAvailable parameter of the Get-Module cmdlet, import the module into the current session before running an Update-Help or Save-Help command. On a remote computer, before running the Save-Help command, import the module into the current CIM or PSSession--or Invoke-Command script block-- that is connected to the remote computer.

When the module is in the current session, run the Update-Help or Save-Help cmdlets without parameters, or use the Module parameter to specify the module name.

The Module parameters of the Update-Help and Save-Help cmdlets accept only a module name. They do not accept the path to a module file.

Use this technique to update or save help for any module that is not returned by the ListAvailable parameter of the Get-Module cmdlet, such as a module that is installed in a location that is not listed in the PSModulePath environment variable, or a module that is not well-formed (the module directory does not contain at least one file whose base name is the same as the directory name).

HOW TO SUPPORT UPDATABLE HELP

If you author a module, you can support online help and Updatable Help for your modules. For more information, see "Supporting Updatable Help" and "Supporting Online Help" in the MSDN Library.

Updatable help not available for Windows PowerShell snap-ins or comment-based help.

KEYWORDS

About_Updateable_Help

REMARKS

The Update-Help and Save-Help cmdlets are not supported on Windows Preinstallation Environment (Windows PE).

SEE ALSO

Get-Help
Save-Help
Update-Help
Updatable Help Status Table
(http://go.microsoft.com/fwlink/?LinkID=270007)

TOPIC

about_Variables

SHORT DESCRIPTION

Describes how variables store values that can be used in Windows PowerShell.

LONG DESCRIPTION

You can store all types of values in Windows PowerShell variables. They are typically used to store the results of commands and to store elements that are used in commands and expressions, such as names, paths, settings, and values.

A variable is a unit of memory in which values are stored. In Windows PowerShell, variables are represented by text strings that begin with a dollar sign ($), such as $a, $process, or $my_var.

Variable names are not case-sensitive. Variable names can include spaces and special characters, but these are difficult to use and should be avoided.

There are several different types of variables in Windows PowerShell.

-- User-created variables: User-created variables are created and maintained by the user. By default, the variables that you create at the Windows PowerShell command line exist only while the Windows PowerShell window is open, and they are lost when you close the window. To save a variable, add it to your Windows PowerShell profile. You can also create variables in scripts with global, script, or local scope.

-- Automatic variables: Automatic variables store the state of Windows PowerShell. These variables are created by Windows PowerShell, and Windows PowerShell changes their values as required to maintain their accuracy. Users cannot change the value of these variables. For example, the $PSHome variable stores the path to the Windows PowerShell installation directory.

For more information, a list, and a description of the automatic variables, see about_Automatic_Variables.

-- Preference variables: Preference variables store user preferences for Windows PowerShell. These variables are created by Windows PowerShell and are populated with default values. Users can change the values of these variables. For example, the $MaximumHistoryCount variable determines the maximum number of entries in the session history.

For more information, a list, and a description of the preference variables, see about_Preference_Variables.

WORKING WITH VARIABLES

To create a new variable, use an assignment statement to assign a value to the variable. You do not have to declare the variable before using it. The default value of all variables is $null.

For example:

```
PS> $MyVariable = 1, 2, 3

PS> $path = "C:\Windows\System32"
```

Variables are very useful for storing the results of commands.

For example:

```
PS> $processes = Get-Process

PS> $Today = (Get-Date).date
```

To display the value of a variable, type the variable name, preceded by a dollar sign ($).

For example:

```
PS> $MyVariable
1
2
3

PS> $Today
Thursday, September 03, 2009 12:00:00 AM
```

To change the value of a variable, assign a new value to the variable.

The following examples displays the value of the $MyVariable variable, changes the value of the variable, and then displays the new value.

```
PS> $MyVariable
1
2
3

PS> $MyVariable = "The green cat."

PS> $MyVariable
The green cat.
```

To delete the value of a variable, use the Clear-Variable cmdlet or change the value to $null.

```
PS> Clear-Variable -name MyVariable

-or-

PS> $MyVariable = $null
```

To delete the variable, use the Remove-Variable or Remove-Item cmdlets. (These cmdlets are discussed later in this topic.)

```
PS> remove-variable -name MyVariable

PS> remove-item -path variable:\myvariable
```

To get a list of all of the variables in your Windows PowerShell session, type:

```
get-variable
```

TYPES OF VARIABLES

You can store any type of object in a variable, including integers, strings, arrays, hash tables, and objects that represent processes, services, event logs, and computers.

Windows PowerShell variables are "loosely typed," which means that they are not limited to a particular type of object. A single variable

can even contain a collection (an "array") of different types of objects at the same time.

The data type of a variable, which is a .NET Framework type, is determined by the .NET types of the values of the variable.

For example:

```
PS> $a = 12     (System.Int32)

PS> $a = "Word" (System.String)

PS> $a = 12, "Word" (System.Int32, System.String)

PS> $a = dir C:\Windows\System32  (Files and folders)
```

You can use a type attribute and cast notation to ensure that a variable can contain only objects of the specified type or objects that can be converted to that type. If you try to assign a value of another type, Windows PowerShell tries to convert the value to its type. If it cannot, the assignment statement fails.

To use cast notation, enter a type name, enclosed in brackets, before the variable name (on the left side of the assignment statement). The following example creates an $number variable that can contain only integers, a $words variable that can contain only strings, and a $dates variable that can contain only DateTime objects.

```
PS> [int]$number = 8

PS> $a = "12345" (The string is converted to an integer.)

PS> $a = "Hello"
Cannot convert value "Hello" to type "System.Int32". Error: "Input string was not in a correct format."
At line:1 char:3
+ $a <<<<  = "Hello"
   + CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
   + FullyQualifiedErrorId : RuntimeException


PS> [string]$words = "Hello"

PS> $words = 2   (The integer is converted to a string.)

PS> $words + 10  (The strings are concatenated.)
210
```

PS> [datetime] $dates = "09/12/91" (The string is converted to a DateTime object.)

PS> $dates
Thursday, September 12, 1991 12:00:00 AM

PS> $dates = 10 (The integer is converted to a DateTime object.)
PS> $dates
Monday, January 01, 0001 12:00:00 AM

USING VARIABLES IN COMMANDS AND EXPRESSIONS

To use a variable in a command or expression, type the variable name, preceded by the dollar sign ($).

If the variable name (and dollar sign) are not enclosed in quotation marks, or if they are enclosed in double quotation marks ("), the value of the variable is used in the command or expression.

If the variable name (and dollar sign) are enclosed in single quotation marks ('), the variable name is used in the expression.

For example, the first command gets the value of the $profile variable, which is the path to the Windows PowerShell user profile file in the Windows PowerShell console. The second command opens the file in Notepad, and the third and fourth commands use the name of the variable in an expression.

PS> $profile
C:\Documents and Settings\User01\My Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1

PS> notepad $profile
- or -
PS> notepad "$profile"
C:\Documents and Settings\User01\My Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1

PS> '$profile'
$profile

PS> 'Use the $profile variable.'
Use the $profile variable.

For more information about using quotation marks in Windows PowerShell, see about_Quoting_Rules.

VARIABLE NAMES THAT INCLUDE SPECIAL CHARACTERS

Variable names begin with a dollar sign. They can include alphanumeric characters and special characters. The length of the variable name is limited only by available memory.

Whenever possible, variable names should include only alphanumeric characters and the underscore character (_). Variable names that include spaces and other special characters, are difficult to use and should be avoided.

To create or display a variable name that includes spaces or special characters, enclose the variable name in braces. This directs Windows PowerShell to interpret the characters in the variable name literally.

For example, the following command creates and then displays a variable named "save-items".

```
C:\PS> ${save-items} = "a", "b", "c"
C:\PS> ${save-items}
a
b
c
```

The following command gets the child items in the directory that is represented by the "ProgramFiles(x86)" environment variable.

```
C:\PS> Get-childitem ${env:ProgramFiles(x86)}
```

To refer to a variable name that includes braces, enclose the variable name in braces, and use the backtick (escape) character to escape the braces. For example, to create a variable named "this{value}is" with a value of 1, type:

```
C:\PS> ${this`{value`}is} = 1
C:\PS> ${this`{value`}is}
1
```

VARIABLES AND SCOPE

By default, variables are available only in the scope in which they are created.

For example, a variable that you create in a function is

available only within the function. A variable that you create in a script is available only within the script (unless you dot-source the script, which adds it to the current scope).

You can use a scope modifier to change the default scope of the variable. The following expression creates a variable named "Computers". The variable has a global scope, even when it is created in a script or function.

```
$global:computers = "Server01"
```

For more information, see about_Scopes.

SAVING VARIABLES

Variables that you create are available only in the session in which you create them. They are lost when you close your session.

To create the in every Windows PowerShell session that you start, add the variable to your Windows PowerShell profile.

For example, to change the value of the $VerbosePreference variable in every Windows PowerShell session, add the following command to your Windows PowerShell profile.

```
$VerbosePreference = "Continue"
```

You can add this command to your profile by opening the profile file in a text editor, such as Notepad. For more information about Windows PowerShell profiles, see about_profiles.

THE VARIABLE: DRIVE

Windows PowerShell Variable provider creates a Variable: drive that looks and acts like a file system drive, but it contains the variables in your session and their values.

To change to the variable: drive, type:

```
set-location variable:
```

(or "cd variable:")

To list the items (variables) in the Variable: drive, use the Get-Item or Get-ChildItem cmdlets. For example:

get-childitem variable:
(or "dir" or "ls")

To get the value of a particular variable, use file system notation to specify the name of the drive and the name of the variable. For example, to get the $PSCulture automatic variable, use the following command.

```
get-item variable:\PSCulture

Name                    Value
----                    -----
PSCulture                en-US
```

For more information about the Variable: drive and the Windows PowerShell Variable provider, type "get-help variable".

THE VARIABLE CMDLETS

Windows PowerShell includes a set of cmdlets that are designed to manage variables.

Cmdlet Name	Description
Clear-Variable	Deletes the value of a variable.
Get-Variable	Gets the variables in the current console.
New-Variable	Creates a new variable.
Remove-Variable	Deletes a variable and its value.
Set-Variable	Changes the value of a variable.

To get help for these cmdlets, type: "Get-Help <cmdlet-hame>".

VARIABLE SQUEEZING

Windows PowerShell supports a simplified syntax for showing the contents of variables during variable assignment. To do this, wrap the variable assignment statement in parentheses.

Typically, assigning and then displaying variable output requires two separate Windows PowerShell commands. But you can consolidate the two processes into one statement by using the variable squeezing technique. The following examples show the difference.

```
    # Assign the variable
    $ProcessList = Get-Process;
    # Display the variable's contents
    $ProcessList;

    # Use variable squeezing to assign and output the variable
    ($ProcessList = Get-Process);
```

SEE ALSO

about_Automatic_Variables
about_Environment_Variables
about_Preference_Variables
about_Profiles
about_Quoting_Rules
about_Scopes

TOPIC

about_While

SHORT DESCRIPTION

Describes a language statement that you can use to run a command block based on the results of a conditional test.

LONG DESCRIPTION

The While statement (also known as a While loop) is a language construct for creating a loop that runs commands in a command block as long as a conditional test evaluates to true. The While statement is easier to construct than a For statement because its syntax is less complicated. In addition, it is more flexible than the Foreach statement because you specify a conditional test in the While statement to control how many times the loop runs.

The following shows the While statement syntax:

```
while (<condition>){<statement list>}
```

When you run a While statement, Windows PowerShell evaluates the <condition> section of the statement before entering the <statement list> section. The condition portion of the statement resolves to either true or false. As long as the condition remains true, Windows PowerShell reruns the <statement list> section.

The <statement list> section of the statement contains one or more commands that are run each time the loop is entered or repeated.

For example, the following While statement displays the numbers 1 through 3 if the $val variable has not been created or if the $val variable has been created and initialized to 0.

```
while($val -ne 3)
{
   $val++
   Write-Host $val
}
```

In this example, the condition ($val is not equal to 3) is true while $val = 0, 1, 2. Each time through the loop, $val is incremented by 1 using the ++ unary increment operator ($val++). The last time through the loop, $val = 3. When $val equals 3, the condition statement evaluates to false, and the loop exits.

To conveniently write this command at the Windows PowerShell command prompt, you can enter it in the following way:

```
while($val -ne 3){$val++; Write-Host $val}
```

Notice that the semicolon separates the first command that adds 1 to $val from the second command that writes the value of $val to the console.

SEE ALSO

about_Comparison_Operators
about_Do

about_Foreach
about_For
about_Language_Keywords

TOPIC

about_Wildcards

SHORT DESCRIPTION

Describes how to use wildcard characters in Windows PowerShell.

LONG DESCRIPTION

Wildcard characters represent one or many characters. You can use them to create word patterns in commands. For example, to get all the files in the C:\Techdocs directory that have a .ppt file name extension, type:

```
Get-ChildItem c:\techdocs\*.ppt
```

In this case, the asterisk (*) wildcard character represents any characters that appear before the .ppt file name extension.

Windows PowerShell supports the following wildcard characters.

Wildcard	Description	Example	Match	No match
*	Matches zero or more characters	a*	A, ag, Apple	banana
?	Matches exactly one character in the specified	?n	an, in, on	ran

position

[ ] Matches a range [a-l]ook book, cook, look took
of characters

[ ] Matches specified [bc]ook book, cook hook
characters

You can include multiple wildcard characters in the same word pattern.
For example, to find text files whose names begin with the letters "a"
through "l", type:

```
Get-ChildItem c:\techdocs\[a-l]*.txt
```

Many cmdlets accept wildcard characters in parameter values. The
Help topic for each cmdlet describes which parameters, if any, permit
wildcard characters. For parameters in which wildcard characters are
accepted, their use is case-insensitive.

You can also use wildcard characters in commands and script blocks, such as
to create a word pattern that represents property values. For example, the
following command gets services in which the ServiceType property value
includes "Interactive".

```
Get-Service | Where-Object {$_.ServiceType -like "*Interactive*"}
```

In the following example, wildcard characters are used to find property values
in the conditions of an If statement. In this command, if the Description of a
restore point includes "PowerShell", the command adds the value of the CreationTime
property of the restore point to a log file.

```
$p = Get-ComputerRestorePoint
foreach ($point in $p)
 {if ($point.description -like "*PowerShell*")
    {add-content -path C:\TechDocs\RestoreLog.txt "$($point.CreationTime)"}}
```

SEE ALSO

about_Language_Keywords
about_If
about_Script_Blocks

TOPIC

about_Windows_PowerShell_5.0

SHORT DESCRIPTION

Describes new features that are included in
Windows PowerShell 5.0.

LONG DESCRIPTION

Windows PowerShell 5.0 includes significant new features that extend its use, improve its usability, and allow you to control and manage Windows-based environments more easily and comprehensively.

Windows PowerShell 5.0 is backward-compatible. Cmdlets, providers, modules, snap-ins, scripts, functions, and profiles that were designed for Windows PowerShell 4.0, Windows PowerShell 3.0, and Windows PowerShell 2.0 generally work in Windows PowerShell 5.0 without changes.

Windows PowerShell 5.0 is installed by default on Windows Server Technical Preview and Windows Technical Preview. To install Windows PowerShell 5.0 on Windows Server 2012 R2, Windows 8.1 Enterprise, or Windows 8.1 Pro, download and install Windows Management Framework 5.0 Preview.
(http://www.microsoft.com/download/details.aspx?id=44070)
Be sure to read the download details, and meet all system requirements, before you install Windows Management Framework 5.0 Preview.

You can also read about changes to Windows PowerShell 5.0 in the Microsoft TechNet topic, "What's New in Windows PowerShell."
(http://go.microsoft.com/fwlink/?LinkID=512808)

NEW FEATURES

New features in Windows PowerShell

-- Starting in Windows PowerShell 5.0, you can develop by using classes, by using formal syntax and semantics that are similar to other object-oriented programming languages. Class, Enum, and other keywords have been added to the

Windows PowerShell language to support the new feature. For more information about working with classes, see about_Classes.
-- In collaboration with Microsoft Research, a new cmdlet, ConvertFrom-String, has been added. ConvertFrom-String lets you extract and parse structured objects from the content of text strings. For more information, see ConvertFrom-String.
-- A new module, Microsoft.PowerShell.Archive, includes cmdlets that let you compress files and folders into archive (also known as ZIP) files, extract files from existing ZIP files, and update ZIP files with newer versions of files compressed within them.
-- A new module, OneGet, lets you discover and install software packages on the Internet. The OneGet module is a manager or multiplexer of existing package managers (also called package providers) to unify Windows package management with a single Windows PowerShell interface.
-- A new module, PowerShellGet, lets you find, install, publish, and update modules and DSC resources on the PowerShell Resource Gallery, or on an internal module repository that you can set up by running the Register-PSRepository cmdlet.
-- New-Item, Remove-Item, and Get-ChildItem have been enhanced to support creating and managing symbolic links. The ItemType parameter for New-Item accepts a new value, SymbolicLink. Now you can create symbolic links in a single line by running the New-Item cmdlet.
-- Windows PowerShell transcription has been improved to apply to all hosting applications (such as Windows PowerShell ISE) in addition to the console host (powershell.exe). Transcription options (including enabling a system-wide transcript) can be configured by enabling the Turn on PowerShell Transcription Group Policy setting, found in Administrative Templates/Windows Components/Windows PowerShell.
-- A new detailed script tracing feature lets you enable detailed tracking and analysis of Windows PowerShell scripting use on a system. After you enable detailed script tracing, Windows PowerShell logs all script blocks to the Event Tracing for Windows (ETW) event log, Microsoft-Windows-PowerShell/Operational.
-- Starting in Windows PowerShell 5.0, new Cryptographic Message Syntax cmdlets support encryption and decryption of content by using the IETF standard format for cryptographically protecting messages as documented by RFC5652 (http://tools.ietf.org/html/rfc5652). The Get-CmsMessage, Protect-CmsMessage, and Unprotect-CmsMessage cmdlets have been added to the Microsoft.PowerShell.Security module.
-- New cmdlets in the Microsoft.PowerShell.Utility module, Get-Runspace, Debug-Runspace, Get-RunspaceDebug, Enable-RunspaceDebug, and Disable-RunspaceDebug, let you set debug options on a runspace, and start and stop debugging on a runspace. For debugging arbitrary runspaces—that is, runspaces that are not the default runspace for a Windows PowerShell console or Windows PowerShell ISE session—Windows PowerShell lets you set breakpoints in a script, and have added breakpoints stop the script from running until you can attach a debugger to debug the runspace script. Nested debugging support for arbitrary runspaces has been added to the Windows PowerShell script debugger for runspaces.
-- New cmdlets Enter-PSHostProcess and Exit-PSHostProcess let you debug

Windows PowerShell scripts in processes separate from the current process that is running in the Windows PowerShell console. Run Enter-PSHostProcess to enter, or attach to, a specific process ID, and then run Get-Runspace to return the active runspaces within the process. Run Exit-PSHostProcess to detach from the process when you are finished debugging the script within the process.

-- A new Wait-Debugger cmdlet has been added to the Microsoft.PowerShell.Utility module. You can run Wait-Debugger to stop a script in the debugger before running the next statement in the script.

-- The Windows PowerShell Workflow debugger now supports command or tab completion, and you can debug nested workflow functions. You can now press Ctrl+Break to enter the debugger in a running script, in both local and remote sessions, and in a workflow script.

-- A Debug-Job cmdlet has been added to the Microsoft.PowerShell.Core module to debug running job scripts for Windows PowerShell Workflow, background, and jobs running in remote sessions.

-- A new state, AtBreakpoint, has been added for Windows PowerShell jobs. The AtBreakpoint state applies when a job is running a script that includes set breakpoints, and the script has hit a breakpoint. When a job is stopped at a debug breakpoint, you must debug the job by running the Debug-Job cmdlet.

-- Windows PowerShell 5.0 implements support for multiple versions of a single Windows PowerShell module in the same folder in $PSModulePath. A RequiredVersion property has been added to the ModuleSpecification class to help you get the desired version of a module; this property is mutually-exclusive with the ModuleVersion property. RequiredVersion is now supported as part of the value of the FullyQualifiedName parameter of the Get-Module, Import-Module, and Remove-Module cmdlets.

-- You can now perform module version validation by running the Test-ModuleManifest cmdlet.

-- Results of the Get-Command cmdlet now display a Version column; a new Version property has been added to the CommandInfo class. Get-Command shows commands from multiple versions of the same module. The Version property is also part of derived classes of CmdletInfo: CmdletInfo and ApplicationInfo.

-- A new Get-ItemPropertyValue cmdlet lets you get the value of a property without using dot notation. For example, in older releases of Windows PowerShell, you can run the following command to get the value of the Application Base property of the PowerShellEngine registry key: (Get-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\PowerShell\3\PowerShellEngine -Name ApplicationBase).ApplicationBase. Starting in Windows PowerShell 5.0, you can run Get-ItemPropertyValue -Path HKLM:\SOFTWARE\Microsoft\PowerShell\3\PowerShellEngine -Name ApplicationBase.

-- A new NetworkSwitch module contains cmdlets that enable you to apply switch, virtual LAN (VLAN), and basic Layer 2 network switch port configuration to Windows Server 2012 R2 (and later releases) logo-certified network switches.

-- The FullyQualifiedName parameter has been added to Import-Module and Remove-Module cmdlets, to support storing multiple versions of a single module.

-- Save-Help, Update-Help, Import-PSSession, Export-PSSession, and Get-Command have a new parameter, FullyQualifiedModule, of type ModuleSpecification. Add this parameter to specify a module by its fully qualified name.

-- The value of $PSVersionTable.PSVersion has been updated to 5.0.

New features in Windows PowerShell Desired State Configuration

-- Windows PowerShell language enhancements let you define Windows PowerShell Desired State Configuration (DSC) resources by using classes.
Import-DscResource is now a true dynamic keyword; Windows PowerShell parses the specified module's root module, searching for classes that contain the DscResource attribute. You can now use classes to define DSC resources, in which neither a MOF file nor a DSCResource subfolder in the module folder is required. A Windows PowerShell module file can contain multiple DSC resource classes.

-- A new parameter, ThrottleLimit, has been added to the following cmdlets in the PSDesiredStateConfiguration module. Add the ThrottleLimit parameter to specify the number of target computers or devices on which you want the command to work at the same time.
  -- Get-DscConfiguration
  -- Get-DscConfigurationStatus
  -- Get-DscLocalConfigurationManager
  -- Restore-DscConfiguration
  -- Test-DscConfiguration
  -- Compare-DscConfiguration
  -- Publish-DscConfiguration
  -- Set-DscLocalConfigurationManager
  -- Start-DscConfiguration
  -- Update-DscConfiguration

-- With centralized DSC error reporting, rich error information is not only logged in the event log, but it can be sent to a central location for later analysis. You can use this central location to store DSC configuration errors that have occurred for any server in their environment. After the report server is defined in the meta-configuration, all errors are sent to the report server, and then stored in a database. You can set up this functionality regardless of whether or not a target node is configured to pull configurations from a pull server.

-- Improvements to Windows PowerShell ISE ease DSC resource authoring. You can now do the following.
  -- List all DSC resources within a configuration or node block by entering Ctrl+Space on a blank line within the block.
  -- Automatic completion on resource properties of the enumeration type.
  -- Automatic completion on the DependsOn property of DSC resources, based on other resource instances in the configuration.
  -- Improved tab completion of resource property values.

-- A new DscLocalConfigurationManager attribute designates a configuration block as a meta-configuration, which is used to configure the DSC Local Configuration Manager. This attribute restricts a configuration to containing only items which configure the DSC Local Configuration Manager. During processing, this configuration generates a *.meta.mof file that is then sent to the

appropriate target nodes by running the Set-DscLocalConfigurationManager cmdlet.

-- Partial configurations are now allowed in Windows PowerShell 5.0. You can deliver configuration documents to a node in fragments. For a node to receive multiple fragments of a configuration document, the node's Local Configuration Manager must be first set to specify the expected fragments.

-- Cross-computer synchronization is new in DSC in Windows PowerShell 5.0. By using the built-in WaitFor* resources (WaitForAll, WaitForAny, and WaitForSome), you can now specify dependencies across computers during configuration runs, without external orchestrations. These resources provide node-to-node synchronization by using CIM connections over the WS-Man protocol. A configuration can wait for another computer's specific resource state to change.

-- Just Enough Administration (JEA), a new delegation security feature, leverages DSC and Windows PowerShell constrained runspaces to help secure enterprises from data loss or compromise by employees, whether intentional or unintentional. For more information about JEA, including where you can download the xJEA DSC resource, see Just Enough Administration, Step by Step. (http://ppe.blogs.technet.com/b/privatecloud/archive/2014/05/14/just-enough-administration-step-by-step.aspx)

-- The following new cmdlets have been added to the PSDesiredStateConfiguration module.

-- A new Get-DscConfigurationStatus cmdlet gets high-level information about configuration status from a target node. You can obtain the status of the last, or of all configurations.

-- A new Compare-DscConfiguration cmdlet compares a specified configuration with the actual state of one or more target nodes.

-- A new Publish-DscConfiguration cmdlet copies a configuration MOF file to a target node, but does not apply the configuration. The configuration is applied during the next consistency pass, or when you run the Update-DscConfiguration cmdlet.

-- A new Test-DscConfiguration cmdlet lets you verify that a resulting configuration matches the desired configuration, returning either True if the configuration matches the desired configuration, or False if the actual configuration does not match the desired configuration.

-- A new Update-DscConfiguration cmdlet forces a configuration to be processed. If the Local Configuration Manager is in pull mode, the cmdlet gets the configuration from the pull server before applying it.

New features in Windows PowerShell ISE

-- You can now edit remote Windows PowerShell scripts and files in a local copy of Windows PowerShell ISE, by running Enter-PSSession to start a remote session on the computer that's storing the files you want to edit, and then running PSEdit <path and file name on the remote computer>. This feature eases editing Windows PowerShell files that are stored on the Server Core installation option of Windows Server, where Windows PowerShell ISE cannot run.

-- The Start-Transcript cmdlet is now supported in Windows PowerShell ISE.

-- You can now debug remote scripts in Windows PowerShell ISE.

-- A new menu command, Break All (Ctrl+B), breaks into the debugger for both

local and remotely-running scripts.

New features in Windows PowerShell Web Services (Management OData IIS Extension)
-- Starting in Windows PowerShell 5.0, you can generate a set of Windows PowerShell cmdlets based on the functionality exposed by a given OData endpoint, by running the Export-ODataEndpointProxy cmdlet.

Notable bug fixes in Windows PowerShell 5.0
-- Windows PowerShell 5.0 includes a new COM implementation, which offers significant performance improvements when you are working with COM objects. For a video demonstration of the effect, see Com_Perf_Improvements. (http://1drv.ms/1qu3UPZ)

For more information about Windows PowerShell 5.0, visit the following web sites:

-- Windows PowerShell Scripting website
http://go.microsoft.com/fwlink/?LinkID=106031

-- Windows PowerShell Team Blog:
http://go.microsoft.com/fwlink/?LinkId=143696

-- Windows PowerShell Web Access
http://technet.microsoft.com/library/hh831611.aspx

SEE ALSO
about_Classes
about_Debuggers
about_Desired_State_Configuration
about_Scheduled_Jobs
about_Updatable_Help
Add-Computer
ConvertFrom-String
Debug-Job
Disable-JobTrigger
Enable-JobTrigger
Get-Module
Get-Process
Invoke-RestMethod
New-JobTrigger
Register-ScheduledJob
Remove-Computer
Save-Help
Set-ExecutionPolicy
Set-JobTrigger
Set-ScheduledJob

Update-Help

KEYWORDS
What's New in Windows PowerShell 5.0

TOPIC

about_Windows_PowerShell_ISE

SHORT DESCRIPTION
Describes the features and system requirements of Windows PowerShell Integrated Scripting Environment (ISE).

LONG DESCRIPTION
Windows PowerShell ISE is a graphical host application for Windows PowerShell. In Windows PowerShell ISE, you can run commands and write, test, and debug scripts in a single Windows-based graphical user interface. Its features include Intellisense, multiline editing, tab completion, auto-save, syntax coloring, selective execution, context-sensitive help, Show Command (compose commands in a window) and support for double-byte character sets and right-to-left languages.

Windows PowerShell ISE is an excellent tool for beginners. The Show Command window and New Remote PowerShell Tab guide you through tasks so that you can be successful on the first try. Snippets and error indicators help you learn the Windows PowerShell language as you work.

Advanced users can take advantage of the sophisticated debugging features, add-ons, and the Windows PowerShell ISE object model.

What's New in Windows PowerShell ISE in Windows PowerShell 4.0

Windows PowerShell ISE introduces two new features in Windows PowerShell 4.0.

-- Windows PowerShell ISE now supports both Windows PowerShell Workflow debugging and remote script debugging. For more Information, see about_Debuggers.
-- IntelliSense support has been added for Windows PowerShell Desired State Configuration providers and configurations.

Starting Windows PowerShell ISE

Windows PowerShell ISE is installed, enabled, and ready to use in all

supported versions of Windows.

- In Windows 8.1, Windows 8, Windows Server 2012 R2, and Windows Server 2012, on the Start screen, type PowerShell_ISE, and then click PowerShell_ISE or Windows PowerShell ISE.

- In Windows Server 2012 R2 and Windows Server 2012, in Server Manager, on the Tools menu, click Windows PowerShell ISE.

- In earlier versions of Windows, click Start, All Programs, Accessories, Windows PowerShell, and then click Windows PowerShell ISE.

- In a Windows PowerShell console, Cmd.exe, or the Run or Search box in Windows, type "PowerShell_ise.exe". You can also use the command-line parameters, including the NoProfile switch. For more information, see PowerShell_ISE.exe Console Help (http://go.microsoft.com/fwlink/?LinkId=243055).

Running Interactive Commands

You can run any Windows PowerShell expression or command in Windows PowerShell ISE. You can use cmdlets, providers, snap-ins, and modules as you would use them in the Windows PowerShell console.

You can type or paste interactive commands in the Console pane. To run the commands, you can use buttons, menu items, and keyboard shortcuts.

You can use the multiline editing feature to type or paste several lines of code into the Console pane at once. When you press the UP ARROW key to recall the previous command, all the lines in the command are recalled. When you type commands, press SHIFT+ENTER to make a new blank line appear under the current line.

Viewing Output

The results of commands and scripts are displayed in the Console pane. You can move or copy the results from the Console pane by using keyboard shortcuts or the Copy button on the toolbar, and you can paste the results in the Script pane or Console panes or other programs. To clear the Console pane, click the "Clear Output Pane" button or type one of the following commands:

```
Clear-Host
cls
```

Writing Scripts and Functions

In the Script pane, you can open, compose, edit, and run scripts. The Script pane lets you edit scripts by using buttons and keyboard shortcuts. You can also copy, cut, and paste text between the Script pane and the Console pane.

You can use the selective run feature to run all or part of a script. To run part of a script, select the text you want to run, and then click the Run Selection button or press F8. By default, F8 runs the current line.

Advanced editing features include brace-matching, expand-collapse, line numbers, error indicators, block editing and indenting, rich copy, and case conversion.

Getting Help

Windows PowerShell ISE includes help topics that describe its use. In addition, all installed help files are accessible from the Script and Command panes.

Windows PowerShell ISE also supports context-sensitive help. To get help about a particular cmdlet, provider, or keyword, place the cursor in the name of the item and press F1. To search the help topics, press F1 and type the search term.

To update the help topics on the computer, use the Update Windows PowerShell Help item in the Help menu. This item updates help for the modules in the current session in the current UI culture. It is equivalent to running the Update-Help cmdlet without parameters. To update help for the cmdlets that come with Windows PowerShell, start Windows PowerShell ISE with the "Run as administrator" option.

You can also use the Get-Help, Save-Help, and Update-Help cmdlets in Windows PowerShell ISE, just as you use it in the Windows PowerShell console. However, in Windows PowerShell ISE, the Help function displays the entire help topic, not one page at a time.

Debugging Scripts

You can use the Windows PowerShell ISE debugger to debug a Windows PowerShell script or function. When you debug a script, you can use menu items and shortcut keys to perform many of the same tasks that you would perform in the Windows PowerShell console. For example, to set a line breakpoint in a script, right-click the line of code, and then click Toggle Breakpoint.

As you step through a script while debugging, the debugging highlighter

shows precisely which part of the command is running and automatically opens files that include called functions and scripts.

By default, the Toggle Breakpoint menu item sets a breakpoint on an entire line in a script, but you can set a breakpoint on a variable or command name. You can also set a breakpoint on a command by line and column number, making it easier to debug long pipeline commands.

Often, you can debug syntax errors in a script just by opening the script file in Windows PowerShell ISE. The error indicators identify syntax errors and the outlining features let you collapse parts of the script to focus on trouble spots.

You can also use the Windows PowerShell debugger cmdlets in the Command pane just as you would use them in the console.

Running Remote Commands

The New Remote PowerShell Tab feature makes it easy to establish a persistent user-managed Windows PowerShell session ("PSSession") to the local computer or a remote computer. The command opens a pop-up window that prompts you for a computer name and for the user account that has permission to run commands on the remote computer.

Customizing the View

You can use Windows PowerShell ISE features to move and to resize the Console pane and the Script pane. You can show and hide either pane, and you can change the text size in all the panes.

You can also use the Options window to customize the appearance and operation of Windows PowerShell ISE. In addition, Windows PowerShell ISE has a custom host variable, $psISE, that you can use to customize Windows PowerShell ISE, including adding menus and menu items.

Windows PowerShell ISE Profile

Windows PowerShell ISE has its own Windows PowerShell profile, Microsoft.PowerShellISE_profile.ps1. In this profile, you can store functions, aliases, variables, and commands that you use in Windows PowerShell ISE.

Items in the Windows PowerShell AllHosts profiles (CurrentUser\AllHosts and AllUsers\AllHosts) are also available in Windows PowerShell ISE, just as they are in any Windows PowerShell host program. However, the items

in your Windows PowerShell console profiles are not available in Windows PowerShell ISE.

Instructions for moving and reconfiguring your profiles are available in Windows PowerShell ISE Help and in about_Profiles.

NOTES

Windows PowerShell ISE is an optional Windows Feature that is turned on by default on client and server versions of Windows. To enable and disable Windows PowerShell ISE in client versions of Windows, use Turn Windows Features On or Off in Control Panel. To enable and disable Windows PowerShell ISE in server versions of Windows, use the Add Roles and Features Wizard in Server Manager.

Because Windows PowerShell ISE requires a user interface, it does not work on Server Core installations of Windows Server. However, if you add the Windows PowerShell ISE feature, the installation automatically converts to Server with a GUI.

Windows PowerShell ISE is built on the Windows Presentation Foundation (WPF). If the graphical elements of Windows PowerShell ISE do not render correctly on your system, you might resolve the problem by adding or adjusting the "Disable WPF Hardware acceleration" graphics rendering settings on your system. For more information, see "Graphics Rendering Registry Settings" in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=144711.

SEE ALSO

about_Debuggers
about_Profiles
about_Updatable_Help
Get-Help
Get-IseSnippet
Import-IseSnippet
New-IseSnippet
Save-Help
Show-Command
Update-Help

TOPIC

about_Windows_RT

SHORT DESCRIPTION

Explains limitations of Windows PowerShell 4.0 on Windows RT 8.1.

LONG DESCRIPTION

The Windows RT 8.1 operating system is installed on computers and devices (such as Microsoft Surface 2, on which it is the operating system that ships with the computer) that use Advanced RISC Machine (ARM) processors.

Windows PowerShell 4.0 is included in Windows RT 8.1. All cmdlets, providers, and modules, and most scripts designed for Windows PowerShell 2.0 and later releases run on Windows RT 8.1 without changes.

Because Windows RT 8.1 does not include all Windows features, some Windows PowerShell features work differently or do not work on Windows RT-based devices. The following list explains the differences.

-- Windows PowerShell ISE is not included in and cannot run on Windows RT 8.1. Windows PowerShell ISE requires Windows Presentation Foundation, which is not included in Windows RT 8.1.

-- Windows PowerShell remoting and the WinRM service are disabled by default. To enable remoting, run the Enable-PSRemoting cmdlet. Also, run the Set-Service cmdlet to set the startup type of the WinRM service to Automatic, or Automatic (Delayed Start).

While remoting is disabled, you can use Windows PowerShell remoting to run commands on other computers, but other computers cannot run commands on the Windows RT device. Also, implicit remoting—that is, remoting that is built in to a cmdlet or script, and not explicitly requested with added parameters—does not work in Windows PowerShell running on Windows RT 8.1.

-- Domain-joined computing and Kerberos authentication are
not supported on Windows RT 8.1. You cannot use Windows PowerShell
to add or manage these features.

-- Microsoft .NET Framework classes that are not supported
on Windows RT 8.1 are also not supported by Windows PowerShell
on Windows RT 8.1.

-- Transactions are not enabled on Windows RT 8.1. Transaction
cmdlets, such as Start-Transaction, and transaction
parameters, such as UseTransaction, do not work properly.

-- All Windows PowerShell sessions on Windows RT 8.1 devices use
the ConstrainedLanguage language mode. ConstrainedLanguage language
mode is a companion to User Mode Code Integrity (UMCI). It permits
all Windows cmdlets and Windows PowerShell language elements, but
restricts types to ensure that users cannot use Windows PowerShell
to circumvent or violate the UMCI protections.

For more information about ConstrainedLanguage language mode, see
about_Language_Modes.

KEYWORDS
about_ARM
about_PowerShell_on_ARM
about_PowerShell_on_Surface
about_Windows_RT_8.1
about_WindowsRT

SEE ALSO
about_Language_Modes
about_Remote
about_Windows_PowerShell_ISE
about_Workflows
Windows PowerShell System Requirements:
(http://technet.microsoft.com/library/hh857337.aspx)

TOPIC

about_WMI

SHORT DESCRIPTION

Windows Management Instrumentation (WMI) uses the Common Information Model (CIM) to represent systems, applications, networks, devices, and other manageable components of the modern enterprise.

LONG DESCRIPTION

Windows Management Instrumentation (WMI) is Microsoft's implementation of Web-Based Enterprise Management (WBEM), the industry standard.

Classic WMI uses DCOM to communicate with networked devices to manage remote systems. Windows PowerShell 3.0 introduces a CIM provider model that uses WinRM to remove the dependency on DCOM. This CIM provider model also uses new WMI provider APIs that enable developers to write Windows PowerShell cmdlets in native code (C++).

Do not confuse WMI providers with Windows PowerShell providers. Many Windows features have an associated WMI provider that exposes their management capabilities. To get WMI providers, run a WMI query that gets instances of the __Provider WMI class, such as the following query.

```
Get-WmiObject -Class __Provider
```

THREE COMPONENTS OF WMI

The following three components of WMI interact with Windows PowerShell: Namespaces, Providers, and Classes.

WMI Namespaces organize WMI providers and WMI classes into groups of related components. In this way, they are similar to .NET Framework namespaces. Namespaces are not physical locations, but are more like logical databases. All WMI namespaces are instances of the __Namespace system class. The default WMI namespace is Root/CIMV2 (since Microsoft Windows 2000). To use Windows PowerShell to get WMI namespaces in the

current session, use a command with the following format.

```
Get-WmiObject -Class __Namespace
```

To get WMI namespaces in other namespaces, use the Namespace parameter to change the location of the search. The following command finds WMI namespaces that reside in the Root/Cimv2/Applications namespace.

```
Get-WmiObject -Class __Namespace -Namespace
root/CIMv2/applications
```

WMI namespaces are hierarchical. Therefore, obtaining a list of all namespaces on a particular system requires performing a recursive query starting at the root namespace.

WMI Providers expose information about Windows manageable objects. A provider retrieves data from a component, and passes that data through WMI to a management application, such as Windows PowerShell. Most WMI providers are dynamic providers, which means that they obtain the data dynamically when it is requested through the management application.

FINDING WMI CLASSES

In a default installation of Windows 8, there are more than 1,100 WMI classes in Root/Cimv2. With this many WMI classes, the challenge becomes identifying the appropriate WMI class to use to perform a specific task. Windows PowerShell 3.0 provides two ways to find WMI classes that are related to a specific topic.

For example,to find WMI classes in the root\CIMV2 WMI namespace that are related to disks, you can use a query such as the one shown here.

```
Get-WmiObject -List *disk*
```

To find WMI classes that are related to memory, you might use a query such as the one shown here.

```
Get-WmiObject -List *memory*
```

The CIM cmdlets also provide the ability to discover WMI classes. To do this, use the Get-CIMClass cmdlet.

The command shown here lists WMI classes related
to video.

```
Get-CimClass *video*
```

Tab expansion works when changing WMI namespaces,
and therefore use of tab expansion makes sub-WMI
namespaces easily discoverable. In the following
example, the Get-CimClass cmdlet lists WMI
classes related to power settings. To find it,
type the root/CIMV2/ WMI namespace, and then press
the Tab key several times until the
power namespace appears. Here is the command:

```
Get-CimClass *power* -Namespace root/cimv2/power
```

TOPIC

about_WMI_cmdlets

SHORT DESCRIPTION

Provides background information about Windows Management Instrumentation
(WMI) and Windows PowerShell.

LONG DESCRIPTION

This topic provides information about WMI technology, the WMI cmdlets for
Windows PowerShell, WMI-based remoting, WMI accelerators,
and WMI troubleshooting. This topic also provides links to more information
about WMI.

About WMI

Windows Management Instrumentation (WMI) is the Microsoft implementation

of Web-Based Enterprise Management (WBEM), which is an industry initiative to develop a standard technology for accessing management information in an enterprise environment. WMI uses the Common Information Model (CIM) industry standard to represent systems, applications, networks, devices, and other managed components. CIM is developed and maintained by the Distributed Management Task Force (DMTF). You can use WMI to manage both local and remote computers. For example, you can use WMI to do the following:

-- Start a process on a remote computer.

-- Restart a computer remotely.

-- Get a list of the applications that are installed on a local or remote computer.

-- Query the Windows event logs on a local or remote computer.

The WMI Cmdlets for Windows PowerShell

Windows PowerShell implements WMI functionality through a set of cmdlets that are available in Windows PowerShell by default. You can use these cmdlets to complete the end-to-end tasks necessary to manage local and remote computers.

The following WMI cmdlets are included.

Cmdlet	Description
Get-WmiObject	Gets instances of WMI classes or information about the available classes.
Invoke-WmiMethod	Calls WMI methods.
Register-WmiEvent	Subscribes to a WMI event.
Remove-WmiObject	Deletes WMI classes and instances.
Set-WmiInstance	Creates or modifies instances of WMI classes.

Sample Commands

The following command displays the BIOS information for the local computer.

```
C:\PS> get-wmiobject win32_bios | format-list *
```

The following command displays information about the WinRM service for three remote computers.

```
C:\PS> get-wmiobject -query "select * from win32_service where name='WinRM'" -computername server01, server01, server03
```

The following more complex command exits all instances of a program.

```
C:\PS> notepad.exe
C:\PS> $np = get-wmiobject -query "select * from win32_process where name='notepad.exe'"
C:\PS> $np | remove-wmiobject
```

WMI-Based Remoting

While the ability to manage a local system through WMI is useful, it is the remoting capabilities that make WMI a powerful administrative tool. WMI uses Microsoft's Distributed Component Object Model (DCOM) to connect to and manage systems. You might have to configure some systems to allow DCOM connections. Firewall settings and locked-down DCOM permissions can block WMI's ability to remotely manage systems.

WMI Type Accelerators

Windows PowerShell includes WMI type accelerators. These WMI type accelerators (shortcuts) allow more direct access to a WMI objects than a non-type accelerator approach would allow.

The following type accelerators are supported with WMI:

[WMISEARCHER] - A shortcut for searching for WMI objects.

[WMICLASS] - A shortcut for accessing the static properties and methods of a class.

[WMI] - A shortcut for getting a single instance of a class.

[WMISEARCHER] is a type accelerator for a ManagementObjectSearcher. It can take a string constructor to create a searcher that you can then do a GET() on.

For example:

```
PS> $s = [WmiSearcher]'Select * from Win32_Process where Handlecount > 1000'
```

```
PS> $s.Get() |sort handlecount |ft handlecount,__path,name -auto

handlecount  __PATH                                    name
-----------  ------                                    ----
1105         \\SERVER01\root\cimv2:Win32_Process.Handle="3724"   PowerShell...
1132         \\SERVER01\root\cimv2:Win32_Process.Handle="1388"   winlogon.exe
1495         \\SERVER01\root\cimv2:Win32_Process.Handle="2852"   iexplore.exe
1699         \\SERVER01\root\cimv2:Win32_Process.Handle="1204"   OUTLOOK.EXE
1719         \\SERVER01\root\cimv2:Win32_Process.Handle="1912"   iexplore.exe
2579         \\SERVER01\root\cimv2:Win32_Process.Handle="1768"   svchost.exe
```

[WMICLASS] is a type accelerator for ManagementClass. This has a string constructor that takes a local or absolute WMI path to a WMI class and returns an object that is bound to that class.

For example:

```
PS> $c = [WMICLASS]"root\cimv2:WIn32_Process"
PS> $c |fl *
Name           : Win32_Process
__GENUS        : 1
__CLASS        : Win32_Process
__SUPERCLASS   : CIM_Process
__DYNASTY      : CIM_ManagedSystemElement
__RELPATH      : Win32_Process
__PROPERTY_COUNT : 45
__DERIVATION   : {CIM_Process, CIM_LogicalElement, CIM_ManagedSystemElement}
__SERVER       : SERVER01
__NAMESPACE    : ROOT\cimv2
__PATH         : \\SERVER01\ROOT\cimv2:Win32_Process
```

[WMI] is a type accelerator for ManagementObject. This has a string constructor that takes a local or absolute WMI path to a WMI instance and returns an object that is bound to that instance.

For example:

```
PS> $p = [WMI]'\\SERVER01\root\cimv2:Win32_Process.Handle="1204"'
PS> $p.Name
OUTLOOK.EXE
```

WMI Troubleshooting

The following problems are the most common problems that might occur when you try to connect to a remote computer.

Problem 1: The remote computer is not online.

If a computer is offline, you will not be able to connect to it by using WMI. You may receive the following error message:

"Remote server machine does not exist or is unavailable"

If you receive this error message, verify that the computer is online. Try to ping the remote computer.

Problem 2: You do not have local administrator rights on the remote computer.

To use WMI remotely, you must have local administrator rights on the remote computer. If you do not, access to that computer will be denied.

To verify namespace security:

a. Click Start, right-click My Computer, and then click Manage.

b. In Computer Management, expand Services and Applications, right-click WMI Control, and then click Properties.

c. In the WMI Control Properties dialog box, click the Security tab.

Problem 3: A firewall is blocking access to the remote computer.

WMI uses the DCOM (Distributed COM) and RPC (Remote Procedure Call) protocols to traverse the network. By default, many firewalls block DCOM and RPC traffic. If your firewall is blocking these protocols, your connection will fail. For example, Windows Firewall in Microsoft Windows XP Service Pack 2 is configured to automatically block all unsolicited network traffic, including DCOM and WMI. In its default configuration, Windows Firewall rejects an incoming WMI request, and you receive the following error message:

"Remote server machine does not exist or is unavailable"

More Information about WMI

For more information about WMI, see the following topics in the MSDN (Microsoft Developer Network) library:

"About WMI:

http://go.microsoft.com/fwlink/?LinkId=142212

"WMI Troubleshooting"
http://go.microsoft.com/fwlink/?LinkId=142213

And, see "Secrets of Windows Management Instrumentation - Troubleshooting and Tips" in the Microsoft TechNet Script Center:

http://go.microsoft.com/fwlink/?LinkId=142214

SEE ALSO
Online version: http://go.microsoft.com/fwlink/?LinkId=142219
Get-WmiObject
Invoke-WmiMethod
Register-WmiEvent
Remove-WmiObject
Set-WmiInstance

Name	Category	Module	Synopsis
about_WorkflowCommonParameters	HelpFile		This topic describes the parameters that are valid on all Windows
about_WorkflowCommonParameters	HelpFile		This topic describes the parameters that are valid on all Windows

Name	Category	Module	Synopsis
about_Workflows	HelpFile		Provides a brief introduction to the Windows
about_Workflows	HelpFile		Provides a brief introduction to the Windows

TOPIC
about_WQL

SHORT DESCRIPTION
Describes WMI Query Language (WQL), which can be used to get WMI objects in Windows PowerShell.

LONG DESCRIPTION
WQL is the Windows Management Instrumentation (WMI) query language, which is the language used to get information from WMI.

You are not required to use WQL to perform a WMI query in Windows PowerShell. Instead, you can use the parameters of the Get-WmiObject or Get-CimInstance cmdlets. WQL queries are somewhat faster than standard Get-WmiObject commands and the improved performance is evident when the commands run on hundreds of systems. However, be sure that the time you spend to write a successful WQL query doesn't outweigh the performance improvement.

The basic WQL statements you need to use WQL are Select, Where, and From.

WHEN TO USE WQL

When working with WMI, and especially with WQL, do not forget that you are also using Windows PowerShell. Often, if a WQL query does not work as expected, it's easier to use a standard Windows PowerShell command than to debug the WQL query.

Unless you are returning massive amounts of data from across bandwidth-constrained remote systems, it is rarely productive to spend hours trying to perfect a complicated and convoluted WQL query when there is a perfectly acceptable Windows cmdlet that does the same thing, if a bit more slowly.

USING THE SELECT STATEMENT

A typical WMI query begins with a Select statement

that gets all properties or particular properties of a WMI class. To select all properties of a WMI class, use an asterisk (*). The From keyword specifies the WMI class.

A Select statement has the following format:

```
Select <property> from <WMI-class>
```

For example, the following Select statement selects all properties (*) from the instances of the Win32_Bios WMI class.

```
Select * from Win32_Bios
```

To select a particular property of a WMI class, place the property name between the Select and From keywords.

The following query selects only the name of the BIOS from the Win32_Bios WMI class. The command saves the query in the $queryName variable.

```
Select Name from Win32_Bios
```

To select more than one property, use commas to separate the property names. The following WMI query selects the name and the version of the Win32_Bios WMI class. The command saves the query in the $queryNameVersion variable.

```
Select name, version from Win32_Bios
```

USING THE WQL QUERY

There are two ways to use WQL query in Windows PowerShell command.

-- Use the Get-WmiObject cmdlet
-- Use the Get-CimInstance cmdlet
-- Use the [wmisearcher] type accelerator.

USING THE GET-WMIOBJECT CMDLET

The most basic way to use the WQL query is to enclose it in quotation marks (as a string) and then use the query string as the value of the Query parameter of

the Get-WmiObject cmdlet, as shown in the following example.

```
PS C:\> Get-WmiObject -Query "Select * from Win32_Bios"

SMBIOSBIOSVersion : 8BET56WW (1.36 )
Manufacturer      : LENOVO
Name              : Default System BIOS
SerialNumber      : R9FPY3P
Version           : LENOVO – 1360
```

You can also save the WQL statement in a variable and then use the variable as the value of the Query parameter, as shown in the following command.

```
PS C:\> $query = "Select * from Win32_Bios"
PS C:\> Get-WmiObject –Query $query
```

You can use either format with any WQL statement. The following command uses the query in the $queryName variable to get only the name and version properties of the system BIOS.

```
PS C:\> $queryNameVersion = "Select Name, Version from Win32_Bios"
PS C:\> Get-WmiObject -Query $queryNameVersion

__GENUS          : 2
__CLASS          : Win32_BIOS
__SUPERCLASS     :
__DYNASTY        :
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
Name             : Default System BIOS
Version          : LENOVO - 1360
```

Remember that you can use the parameters of the Get-WmiObject cmdlet to get the same result. For example, the following command also gets the values of the Name and Version properties of instances of the Win32_Bios WMI class.

```
PS C:\> Get-WmiObject –Class Win32_Bios -Property Name, Version
```

```
__GENUS          : 2
__CLASS          : Win32_BIOS
__SUPERCLASS     :
__DYNASTY        :
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
Name             : Default System BIOS
Version          : LENOVO - 1360
```

USING THE GET-CIMINSTANCE CMDLET

Beginning in Windows PowerShell 3.0, you can use the Get-CimInstance cmdlet to run WQL queries.

Get-CimInstance gets instances of CIM-compliant classes, including WMI classes. The CIM cmdlets, introduced Windows PowerShell 3.0, perform the same tasks as the WMI cmdlets. The CIM cmdlets comply with WS-Management (WSMan) standards and with the Common Information Model (CIM) standard, which enables the cmdlets to use the same techniques to manage Windows computers and computers that are running other operating systems.

The following command uses the Get-CimInstance cmdlet to run a WQL query.

Any WQL query that can be used with Get-WmiObject can also be used with Get-CimInstance.

```
PS C:\> Get-CimInstance -Query "Select * from Win32_Bios"

SMBIOSBIOSVersion : 8BET56WW (1.36 )
Manufacturer      : LENOVO
Name              : Default System BIOS
SerialNumber      : R9FPY3P
Version           : LENOVO – 1360
```

Get-CimInstance returns a CimInstance object, instead of the ManagementObject that Get-WmiObject returns, but the objects are quite similar.

```
PS C:\>(Get-CimInstance -Query "Select * from Win32_Bios").GetType().FullName
Microsoft.Management.Infrastructure.CimInstance
PS C:\>(Get-WmiObject -Query "Select * from Win32_Bios").GetType().FullName
System.Management.ManagementObject
```

USING THE [wmisearcher] TYPE ACCELERATOR

The [wmisearcher] type accelerator creates a ManagementObjectSearcher object from a WQL statement string. The ManagementObjectSearcher object has many properties and methods, but the most basic method is the Get method, which invokes the specified WMI query and returns the resulting objects.

By using the [wmisearcher], you gain easy access to the ManagementObjectSearcher .NET Framework class. This lets you query WMI and to configure the way the query is conducted.

To use the [wmisearcher] type accelerator:

1. Cast the WQL string into a ManagementObjectSearcher object.
2. Call the Get method of the ManagementObjectSearcher object.

For example, the following command casts the "select all" query, saves the result in the $bios variable, and then calls the Get method of the ManagementObjectSearcher object in the $bios variable.

```
PS C:\> $bios = [wmisearcher]"Select * from Win32_Bios"
PS C:\> $bios.Get()

SMBIOSBIOSVersion : 8BET56WW (1.36 )
Manufacturer      : LENOVO
Name              : Default System BIOS
SerialNumber      : R9FPY3P
Version           : LENOVO – 1360
```

NOTE: Only selected object properties are displayed by default. These properties are defined in the Types.ps1xml file.

You can use the [wmisearcher] type accelerator to cast the query or the variable. In the following example, the [wmisearcher] type accelerator is used to cast the variable. The result is the same.

```
PS C:\> [wmisearcher]$bios = "Select * from Win32_Bios"
PS C:\> $bios.Get()

SMBIOSBIOSVersion : 8BET56WW (1.36 )
Manufacturer      : LENOVO
Name              : Default System BIOS
SerialNumber      : R9FPY3P
Version           : LENOVO – 1360
```

When you use the [wmisearcher] type accelerator, it changes the query string into a ManagementObjectSearcher object, as shown in the following commands.

```
PS C:\>$a = "Select * from Win32_Bios"
PS C:\>$a.GetType().FullName
System.String

PS C:\>$a = [wmisearcher]"Select * from Win32_Bios"
PS C:\>$a.GetType().FullName
System.Management.ManagementObjectSearcher
```

This command format works on any query. The following command gets the value of the Name property of the Win32_Bios WMI class.

```
PS C:\> $biosname = [wmisearcher]"Select Name from Win32_Bios"
PS C:\> $biosname.Get()

__GENUS          : 2
__CLASS          : Win32_BIOS
__SUPERCLASS     :
__DYNASTY        :
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
Name             : Default System BIOS
```

You can perform this operation in a single command, although the command is a bit more difficult to interpret.

In this format, you use the [wmisearcher] type accelerator to cast the WQL query string to a ManagementObjectSearcher, and then call the Get method on the object -- all in a single command. The parentheses () that enclose the casted string direct Windows PowerShell to cast the string before calling the method.

```
PS C:\> ([wmisearcher]"Select name from Win32_Bios").Get()

__GENUS          : 2
__CLASS          : Win32_BIOS
__SUPERCLASS     :
__DYNASTY        :
__RELPATH        :
__PROPERTY_COUNT : 1
__DERIVATION     : {}
__SERVER         :
__NAMESPACE      :
__PATH           :
Name             : Default System BIOS
```

USING THE BASIC WQL WHERE STATEMENT

A Where statement establishes conditions for the data that a Select statement returns.

The Where statement has the following format:

```
where <property> <operator> <value>
```

For example:

```
where Name = 'Notepad.exe'
```

The Where statement is used with the Select statement, as shown in the following example.

```
Select * from Win32_Process where Name = 'Notepad.exe'
```

When using the Where statement, the property name

and value must be accurate.

For example, the following command gets the Notepad processes on the local computer.

```
PS C:\> Get-WmiObject -Query "Select * from Win32_Process where name = 'Notepad.exe'"
```

However, the following command fails, because the process name includes the ".exe" file name extension.

```
PS C:\> Get-WmiObject -Query "Select * from Win32_Process where name = 'Notepad'"
```

WHERE STATEMENT COMPARISON OPERATORS

The following operators are valid in a WQL Where statement.

Operator	Description
=	Equal
!=	Not equal
<>	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
LIKE	Wildcard match
IS	Evaluates null
ISNOT	Evaluates not null
ISA	Evaluates a member of a WMI class

There are other operators, but these are the ones used for making comparisons.

For example, the following query selects the Name and Priority properties from processes in the Win32_Process class where the process priority is greater than or equal to 11. The Get-WmiObject cmdlet runs the query.

```
$highPriority = "Select Name, Priority from Win32_Process where Priority >= 11"
Get-WmiObject -Query $highPriority
```

USING THE WQL OPERATORS IN THE FILTER PARAMETER

The WQL operators can also be used in the value

of the Filter parameter of the Get-WmiObject or Get-CimInstance cmdlets, as well as in the value of the Query parameters of these cmdlets.

For example, the following command gets the Name and ProcessID properties of the last five processes that have ProcessID values greater than 1004. The command uses the Filter parameter to specify the ProcessID condition.

```
PS C:\> Get-WmiObject -Class Win32_Process `
-Property Name, ProcessID -Filter "ProcessID >= 1004" |
Sort ProcessID | Select Name, ProcessID -Last 5

Name                    ProcessID
----                    ---------
SROSVC.exe                 4220
WINWORD.EXE                4664
TscHelp.exe                4744
SnagIt32.exe               4748
WmiPrvSE.exe               5056
```

USING THE LIKE OPERATOR

The Like operator lets you use wildcard characters to filter the results of a WQL query.

Like Operator	Description
[]	Character in a range [a-f] or a set of characters [abcdef]. The items in a set do not need to be consecutive or listed in alphabetical order.
^	Character not in a range [^a-f] or not in a set [^abcdef]. The items in a set do not need to be consecutive or listed in alphabetical order.
%	A string of zero or more characters
_ (underscore)	One character. NOTE: To use a literal underscore in a query string, enclose it in square brackets [_].

When the Like operator is used without any wildcard

characters or range operators, it behaves like the
equality operator (=) and returns objects only
when they are an exact match for the pattern.

You can combine the range operation with the percent
wildcard character to create simple, yet powerful
filters.

LIKE OPERATOR EXAMPLES

EXAMPLE 1: [<range>]
The following commands start Notepad and then
search for an instance of the Win32_Process
class that has a name that starts with a letter
between "H" and "N" (case-insensitive).

The query should return any process from Hotpad.exe
through Notepad.exe.

```
PS C:\> Notepad   # Starts Notepad
PS C:\> $query = "Select * from win32_Process where Name LIKE '[H-N]otepad.exe'"
PS C:\> Get-WmiObject -Query $query | Select Name, ProcessID

Name                      ProcessID
----                      ---------
notepad.exe                    1740
```

EXAMPLE 2: [<range>] and %
The following commands select all process
that have a name that begins with a letter between
A and P (case-insensitive) followed by zero or more
letters in any combination.

The Get-WmiObject cmdlet runs the query, the
Select-Object cmdlet gets the Name and ProcessID
properties, and the Sort-Object cmdlet sorts the
results in alphabetical order by name.

```
PS C:\>$query = "Select * from win32_Process where name LIKE '[A-P]%'"
PS C:\>Get-WmiObject -Query $query |
        Select-Object -Property Name, ProcessID |
        Sort-Object -Property Name
```

EXAMPLE 3: Not in Range (^)
The following command gets processes whose names

do not begin with any of the following letters:
A, S, W, P, R, C, U, N
and followed zero or more letters.

```
PS C:\>$query = "Select * from win32_Process where name LIKE '[^ASWPRCUN]%'"
PS C:\>Get-WmiObject -Query $query |
        Select-Object -Property Name, ProcessID |
        Sort-Object -Property Name
```

EXAMPLE 4: Any characters -- or none (%)
The following commands get processes that have names that begin with "calc". The % symbol in WQL is equivalent to the asterisk (*) symbol in regular expressions.

```
PS C:\> $query = "Select * from win32_Process where Name LIKE 'calc%'"
PS C:\> Get-WmiObject -Query $query | Select-Object -Property Name, ProcessID

Name                      ProcessID
----                      ---------
calc.exe                     4424
```

EXAMPLE 5: One character (_)
The following commands get processes that have names that have the following pattern, "c_lc.exe" where the underscore character represents any one character. This pattern matches any name from calc.exe through czlc.exe, or c9lc.exe, but does not match names in which the "c" and "l" are separated by more than one character.

```
PS C:\> $query = "Select * from Win32_Process where Name LIKE 'c_lc.exe'"
PS C:\> Get-WmiObject -Query $query | Select-Object -Property Name, ProcessID

Name                      ProcessID
----                      ---------
calc.exe                     4424
```

EXAMPLE 6: Exact match
The following commands get processes named WLIDSVC.exe. Even though the query uses the Like keyword, it requires an exact match, because the value does not include any wildcard characters.

```
$query = "Select * from win32_Process where name LIKE 'WLIDSVC.exe'"
```

```
Get-WmiObject -Query $query | Select-Object -Property Name, ProcessID

Name                    ProcessID
----                    ---------
WLIDSVC.exe                    84
```

USING THE OR OPERATOR

To specify multiple independent conditions, use the Or keyword. The Or keyword appears in the Where clause. It performs an inclusive OR operation on two (or more) conditions and returns items that meet any of the conditions.

The Or operator has the following format:

```
Where <property> <operator> <value> or <property> <operator> <value> ...
```

For example, the following commands get all instances of the Win32_Process WMI class but returns them only if the process name is winword.exe or excel.exe.

```
PS C:\>$q = "Select * from Win32_Process where Name = 'winword.exe' or Name = 'excel.exe'"
PS C:\>Get-WmiObject -Query $q
```

The Or statement can be used with more than two conditions. In the following query, the Or statement gets Winword.exe, Excel.exe, or Powershell.exe.

```
$q = "Select * from Win32_Process where Name = 'winword.exe' or Name = 'excel.exe' or Name = 'powershell.exe'"
```

USING THE AND OPERATOR

To specify multiple related conditions, use the And keyword. The And keyword appears in the Where clause. It returns items that meet all of the conditions.

The And operator has the following format:

```
Where <property> <operator> <value> and <property> <operator> <value> ...
```

For example, the following commands get processes that have a name of "Winword.exe" and the process ID of 6512.

Note that the commands use the Get-CimInstance cmdlet.

```
PS C:\>$q = "Select * from Win32_Process where Name = 'winword.exe' and ProcessID =6512"
PS C:\> Get-CimInstance -Query $q

ProcessId      Name          HandleCount     WorkingSetSize   VirtualSize
---------      ----          -----------     --------------   -----------
6512           WINWORD.EXE   768             117170176        633028608
```

All operators, including the Like operators are valid with the Or and And operators. And, you can combine the Or and And operators in a single query with parentheses that tell Windows PowerShell which clauses to process first.

This command uses the Windows PowerShell continuation character (`) divide the command into two lines.

```
PS C:\> $q = "Select * from Win32_Process `
where (Name = 'winword.exe' or Name = 'excel.exe') and HandleCount > 700"

PS C:\> Get-CimInstance -Query $q
ProcessId      Name          HandleCount     WorkingSetSize   VirtualSize
---------      ----          -----------     --------------   -----------
6512           WINWORD.EXE   797             117268480        634425344
9610           EXCEL.EXE     727             38858752         323227648
```

SEARCHING FOR NULL VALUES

Searching for null values in WMI is challenging, because it can lead to unpredictable results. Null is not zero and it is not equivalent or to an empty string. Some WMI class properties are initialized and others are not, so a search for null might not work for all properties.

To search for null values, use the Is operator with a value of "null".

For example, the following commands get processes that have a null value for the IntallDate property. The commands return many processes.

```
PS C:\>$q = "Select * from Win32_Process where InstallDate is null"
PS C:\>Get-WmiObject -Query $q
```

In contrast, the following command, gets user accounts that have a null value for the Description

property. This command does not return any user accounts, even though most user accounts do not have any value for the Description property.

```
PS C:\>$q = "Select * from Win32_UserAccount where Description is null"
PS C:\>Get-WmiObject -Query $q
```

To find the user accounts that have no value for the Description property, use the equality operator to get an empty string. To represent the empty string, use two consecutive single quotation marks.

```
$q = "Select * from Win32_UserAccount where Description = '' "
```

USING TRUE OR FALSE

To get Boolean values in the properties of WMI objects, use True and False. They are not case sensitive.

The following WQL query returns only local user accounts from a domain joined computer.

```
PS C:\>$q = "Select * from Win32_UserAccount where LocalAccount = True"
PS C:\>Get-CimInstance -Query $q
```

To find domain accounts, use a value of False, as shown in the following example.

```
PS C:\>$q = "Select * from Win32_UserAccount where LocalAccount = False"
PS C:\>Get-CimInstance -Query $q
```

USING THE ESCAPE CHARACTER

WQL uses the backslash (\) as its escape character. This is different from Windows PowerShell, which uses the backtick character (`).

Quotation marks, and the characters used for quotation marks, often need to be escaped so that they are not misinterpreted.

To find a user whose name includes a single quotation mark, use a backslash to escape the single quotation mark, as shown in the following command.

```
PS C:\> $q = "Select * from Win32_UserAccount where Name = 'Tim O\'Brian'"
```

```
PS C:\> Get-CimInstance -Query $q
Name          Caption        AccountType    SID           Domain
----          -------        -----------    ---           ------
Tim O'Brian   FABRIKAM\TimO  512            S-1-5-21-1457... FABRIKAM
```

In some case, the backslash also needs to be escaped. For example, the following commands generate an Invalid Query error due to the backslash in the Caption value.

```
PS C:\> $q = "Select * from Win32_UserAccount where Caption = 'Fabrikam\TimO'"
PS C:\> Get-CimInstance -Query $q
Get-CimInstance : Invalid query
At line:1 char:1
+ Get-CimInstance -Query $q
+ ~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-CimInstance], CimException
    + FullyQualifiedErrorId : HRESULT 0x80041017,Microsoft.Management.Infrastructure.CimCmdlets
```

To escape the backslash, use a second backslash character, as shown in the following command.

```
PS C:\> $q = "Select * from Win32_UserAccount where Caption = 'Fabrikam\\TimO'"
PS C:\> Get-CimInstance -Query $q
```

SEE ALSO

about_Escape_Characters
about_Quoting_Rules
about_WMI
about_WMI_Cmdlets

TOPIC

about_WS-Management_Cmdlets

SHORT DESCRIPTION

Provides an overview of Web Services for Management (WS-Management) as background for using the WS-Management cmdlets in Windows PowerShell.

LONG DESCRIPTION

This topic provides an overview of Web Services for Management (WS-Management) as background for using the WS-Management cmdlets in Windows PowerShell. This topic also provides links to more information about WS-Management. The Microsoft implementation of WS-Management is also known as Windows Remote Management (WinRM).

About WS-Management

Windows Remote Management is the Microsoft implementation of the WS-Management protocol, a standard SOAP-based, firewall-friendly protocol that allows hardware and operating systems from different vendors to interoperate. The WS-Management protocol specification provides a common way for systems to access and exchange management information across an information technology (IT) infrastructure. WS-Management and Intelligent Platform Management Interface (IPMI), along with the Event Collector, are components of the Windows Hardware Management features.

The WS-Management protocol is based on the following standard Web service specifications: HTTPS, SOAP over HTTP (WS-I profile), SOAP 1.2, WS-Addressing, WS-Transfer, WS-Enumeration, and WS-Eventing.

WS-Management and WMI

WS-Management can be used to retrieve data exposed by Windows Management Instrumentation (WMI). You can obtain WMI data with scripts or applications that use the WS-Management Scripting API or through the WinRM command-line tool. WS-Management supports most of the familiar WMI classes and operations, including embedded objects. WS-Management can

leverage WMI to collect data about resources or to manage resources on a Windows-based computers. That means that you can obtain data about objects such as disks, network adapters, services, or processes in your enterprise through the existing set of WMI classes. You can also access the hardware data that is available from the standard WMI IPMI provider.

WS-Management Windows PowerShell Provider (WSMan)

The WSMan provider provides a hierarchical view into the available WS-Management configuration settings. The provider allows you to explore and set the various WS-Management configuration options.

WS-Management Configuration

If WS-Management is not installed and configured, Windows PowerShell remoting is not available, the WS-Management cmdlets do not run, WS-Management scripts do not run, and the WSMan provider cannot perform data operations. The WS-Management command-line tool, WinRM, and event forwarding also depend on the WS-Management configuration.

WS-Management Cmdlets

WS-Management functionality is implemented in Windows PowerShell through a module that contains a set of cmdlets and the WSMan provider. You can use these cmdlets to complete the end-to-end tasks necessary to manage WS-Management settings on local and remote computers.

The following WS-Management cmdlets are available.

Connection Cmdlets

-- Connect-WSMan: Connects the local computer to the WS-Management (WinRM) service on a remote computer.

-- Disconnect-WSMan: Disconnects the local computer from the WS-Management (WinRM) service on a remote computer.

Management-Data Cmdlets

-- Get-WSManInstance: Displays management information for a resource instance that is specified by a resource URI.

-- Invoke-WSManAction: Invokes an action on the target object that is specified by the resource URI and by the selectors.

-- New-WSManInstance: Creates a new management resource instance.

-- Remove-WSManInstance: Deletes a management resource instance.

-- Set-WSManInstance: Modifies the management information that is related to a resource.

Setup and Configuration Cmdlets

-- Set-WSManQuickConfig: Configures the local computer for remote management. You can use the Set-WSManQuickConfig cmdlet to configure WS-Management to allow remote connections to the WS-Management (WinRM) service. The Set-WSManQuickConfig cmdlet performs the following operations:

- It determines whether the WS-Management (WinRM) service is running. If the WinRM service is not running, the Set-WSManQuickConfig cmdlet starts the service.

- It sets the WS-Management (WinRM) service startup type to automatic.

- It creates a listener that accepts requests from any IP address. The default transport protocol is HTTP.

- It enables a firewall exception for WS-Management traffic.

Note: To run this cmdlet in Windows Vista, Windows Server 2008, and later versions of Windows, you must start Windows PowerShell with the "Run as administrator" option.

-- Test-WSMan: Verifies that WS-Management is installed and configured. The Test-WSMan cmdlet tests whether the WS-Management (WinRM) service is running and configured on a local or remote computer.

-- Disable-WSManCredSSP: Disables CredSSP authentication on a client computer.

-- Enable-WSManCredSSP: Enables CredSSP authentication on a client computer.

-- Get-WSManCredSSP: Gets the CredSSP-related configuration for a client computer.

WS-Management-Specific Cmdlets

-- New-WSManSessionOption: Creates a WSManSessionOption object to use as input to one or more parameters of a WS-Management cmdlet.

Additional WS-Management Information

For more information about WS-Management, see the following topics in the MSDN (Microsoft Developer Network) library.

"Windows Remote Management"
http://go.microsoft.com/fwlink/?LinkId=142321

"About Windows Remote Management"
http://go.microsoft.com/fwlink/?LinkId=142322

"Installation and Configuration for Windows Remote Management"
http://go.microsoft.com/fwlink/?LinkId=142323

"Windows Remote Management Architecture"
http://go.microsoft.com/fwlink/?LinkId=142324

"WS-Management Protocol"
http://go.microsoft.com/fwlink/?LinkId=142325

"Windows Remote Management and WMI"
http://go.microsoft.com/fwlink/?LinkId=142326

"Resource URIs"
http://go.microsoft.com/fwlink/?LinkId=142327

"Remote Hardware Management"
http://go.microsoft.com/fwlink/?LinkId=142328

"Events"
http://go.microsoft.com/fwlink/?LinkId=142329

SEE ALSO

Online version: http://go.microsoft.com/fwlink/?LinkId=142331
Connect-WSMan
Disable-WSManCredSSP
Disconnect-WSMan
Enable-WSManCredSSP
Get-WSManCredSSP
Get-WSManInstance
Invoke-WSManAction
New-WSManInstance
Remove-WSManInstance
Set-WSManInstance
Set-WSManQuickConfig

Set-WSManSessionOption
Test-WSMan

KEYWORDS
about_WSMAN about_WinRM

about_WS-Management_Cmdlets.help.txt,WSPOWERSHELLv4,about_Aliases

TOPIC

About_MVADemo

SHORT DESCRIPTION
This contains two resources to manage Widnows Features and Services.

EXAMPLES

```
MVAFeature [String] #ResourceName
{
    FeatureName = [string]
    [DependsOn = [string[]]]
    [Ensure = [string]{ Absent | Present }]
    [Installed = [bool]]
}
```

TOPIC

about_BITS_Cmdlets

SHORT DESCRIPTION
Provides background information about the Background Intelligent Transfer Service (BITS).

LONG DESCRIPTION
This topic provides information about BITS, BITS transfer types, BITS transfer persistence, BITS transfer priority, the Windows PowerShell and BITS cmdlets, and BITS server configuration. This topic also provides links to more information about BITS.

About BITS

BITS is a file transfer service that provides a scriptable interface through Windows PowerShell. BITS transfers files asynchronously in the

foreground or in the background. And, it automatically resumes file transfers after network disconnections and after a computer is restarted.

Background transfers use only idle network bandwidth in an effort to preserve the user's interactive experience with other network applications such as Internet Explorer. BITS does this by examining the network traffic and then using only the idle portion of the network bandwidth. BITS continuously throttles its use of the bandwidth as the user increases or decreases their use of the bandwidth. BITS performs the transfers asynchronously, which means that your program or script does not have to be running for BITS to perform the transfer. Both uploads and downloads are supported. BITS is particularly suitable for copying files from an HTTP location in an Internet Information Services virtual directory to the logical drive of a client. Standard server message block (SMB) copy operations are also supported. In addition to the scripting interface provided by Windows PowerShell, BITS provides a set of COM APIs to allow programming access.

BITS Transfer Types

There are three types of BITS transfer jobs:

- A download job downloads files to the client computer.

- An upload job uploads a file to the server.

- An upload-reply job uploads a file to the server and receives a reply file from the server application.

BITS Transfer Persistence

BITS continues to transfer files after an application exits if the user who initiated the transfer remains logged on and if a network connection is maintained. BITS suspends the transfer if a connection is lost or if the user logs off. BITS also persists the transfer information when a user logs off, when network disconnections occur, and when a computer is restarted. When the user logs on again, when the network is reconnected, and when the computer is restarted, BITS resumes the user's transfer job.

BITS Transfer Priority

BITS provides one foreground and three background priority levels that you can use to prioritize transfer jobs. Higher priority jobs preempt lower priority jobs. Jobs at the same priority level share transfer time,

which prevents a large job from blocking small jobs in the transfer queue. Lower priority jobs do not receive transfer time until all the higher priority jobs are complete or in an error state. Background transfers are optimal because BITS uses idle network bandwidth to transfer the files. BITS increases or decreases the rate at which files are transferred based on the amount of idle network bandwidth that is available. If a network application begins to consume more bandwidth, BITS decreases its transfer rate to preserve the user's interactive experience. BITS supports multiple foreground jobs and one background transfer job at the same time.

Windows PowerShell and the BITS Cmdlets

Windows PowerShell implements BITS functionality through the BITS module for Windows PowerShell. The BITS module loads a set of BITS-specific cmdlets. You can use these cmdlets to complete the end-to-end tasks that are necessary to manage the transfer of files between computers.

When the BITS module for Windows PowerShell is loaded, the following BITS cmdlets are available.

Cmdlet	Descriptions
Add-BitsFile	Adds one or more files to a BITS transfer.
Complete-BitsTransfer	Completes a BITS transfer.
Get-BitsTransfer	Gets a single or multiple BITS transfer.
Remove-BitsTransfer	Deletes a BITS transfer.
Resume-BitsTransfer	Resumes a suspended BITS transfer.
Set-BitsTransfer	Configures BITS transfer jobs.
Start-BitsTransfer	Creates and starts a BITS transfer job.
Suspend-BitsTransfer	Suspends a BITS transfer job.

To copy a file using BITS:

1. Create a BITS transfer job by using the Start-BitsTranfer cmdlet, optionally with the Suspend parameter.

2. Add files to the BITS transfer job by using the Add-BitsFile cmdlet.

3. Start the BITS transfer by using the Resume-BitsTransfer cmdlet.

4. Check the status of the BITS transfer job by using the Get-BitsTransfer cmdlet.

Sample Commands

A simple Windows PowerShell BITS file transfer command might resemble the following command:

```
C:\PS> Start-BitsTransfer http://server01/servertestdir/testfile1.txt c:\clienttestdir\testfile1.txt
```

A slightly more complex Windows PowerShell BITS file transfer set of commands might resemble the following command:

```
C:\PS> Import-CSV filelist.txt | Start-BitsTransfer -TransferType Upload
```

BITS Server Configuration

Background Intelligent Transfer Services (BITS) server extends Internet Information Services (IIS) to support throttled uploads that can be restarted. To upload files to a server by using BITS, the server must be running IIS 7.0 and Windows Server 2008. Additionally, the BITS server extension for the Internet Server Application Programming Interface (ISAPI) must be installed. The BITS server extension is a subcomponent of IIS. To use the upload feature, create an IIS virtual directory on the server where clients can upload files. Create a virtual directory for each type of client. BITS adds properties to the IIS metabase for the virtual directory that you create, and it uses these properties to determine how to upload the files. For more information, see "Setting Up the Server for Uploads" in the MSDN library at http://go.microsoft.com/fwlink/?LinkId=142215.

More Information about BITS

For more information about BITS, see the following topics in the MSDN (Microsoft Developer Network) library:

- "Background Intelligent Transfer Service" http://go.microsoft.com/fwlink/?LinkId=142216

- "About BITS" http://go.microsoft.com/fwlink/?LinkId=142217

SEE ALSO

Online version: http://go.microsoft.com/fwlink/?LinkId=142218
Add-BitsFile
Complete-BitsTransfer
Get-BitsTransfer
Remove-BitsTransfer

Resume-BitsTransfer
Set-BitsTransfer
Start-BitsTransfer
Suspend-BitsTransfer

Name	Category	Module	Synopsis
about_Scheduled_Jobs	HelpFile		Describes scheduled jobs and explains how to use and manage
about_Scheduled_Jobs	HelpFile		Describes scheduled jobs and explains how to use and manage

Name	Category	Module	Synopsis
about_Scheduled_Jobs_Advanced	HelpFile		Explains advanced scheduled job topics, including the file structure
about_Scheduled_Jobs_Advanced	HelpFile		Explains advanced scheduled job topics, including the file structure

Name	Category	Module	Synopsis
about_Scheduled_Jobs_Basics	HelpFile		Explains how to create and manage scheduled jobs.
about_Scheduled_Jobs_Basics	HelpFile		Explains how to create and manage scheduled jobs.

Name	Category	Module	Synopsis
about_Scheduled_Jobs_Troublesh...	HelpFile		Explains how to resolve problems with scheduled jobs
about_Scheduled_Jobs_Troublesh...	HelpFile		Explains how to resolve problems with scheduled jobs

Name	Category	Module	Synopsis
about_ActivityCommonParameters	HelpFile		Describes the parameters that Windows PowerShell

about_ActivityCommonParameters HelpFile Describes the parameters that Windows PowerShell

Name Category Module Synopsis
---- -------- ------ --------
about_Checkpoint-Workflow HelpFile Describes the Checkpoint-Workflow activity, which
about_Checkpoint-Workflow HelpFile Describes the Checkpoint-Workflow activity, which

Name Category Module Synopsis
---- -------- ------ --------
about_ForEach-Parallel HelpFile Describes the ForEach -Parallel language construct in
about_ForEach-Parallel HelpFile Describes the ForEach -Parallel language construct in

Name Category Module Synopsis
---- -------- ------ --------
about_InlineScript HelpFile Describes the InlineScript activity, which runs Windows
about_InlineScript HelpFile Describes the InlineScript activity, which runs Windows

Name Category Module Synopsis
---- -------- ------ --------
about_Parallel HelpFile Describes the Parallel keyword, which runs the
about_Parallel HelpFile Describes the Parallel keyword, which runs the

Name Category Module Synopsis
---- -------- ------ --------
about_Sequence HelpFile Describes the Sequence keyword, which runs selected
about_Sequence HelpFile Describes the Sequence keyword, which runs selected

Name	Category	Module	Synopsis
about_Suspend-Workflow	HelpFile		Describes the Suspend-Workflow activity, which suspends
about_Suspend-Workflow	HelpFile		Describes the Suspend-Workflow activity, which suspends

Name	Category	Module	Synopsis
about_WorkflowCommonParameters	HelpFile		This topic describes the parameters that are valid on all Windows
about_WorkflowCommonParameters	HelpFile		This topic describes the parameters that are valid on all Windows

Name	Category	Module	Synopsis
about_Workflows	HelpFile		Provides a brief introduction to the Windows
about_Workflows	HelpFile		Provides a brief introduction to the Windows