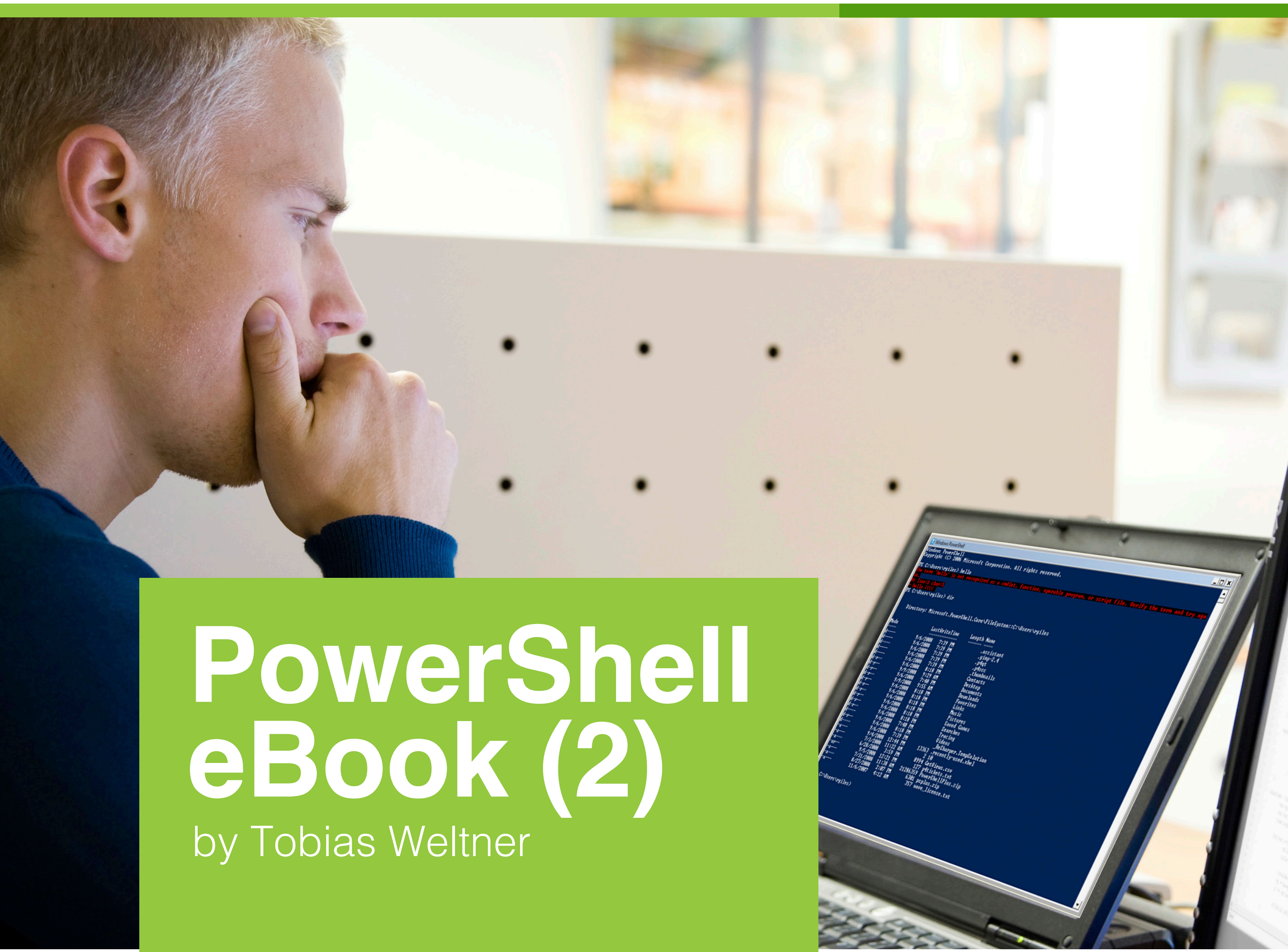


PowerShell eBook (2)

by Tobias Weltner

Index

by Tobias Weltner

Chapter 7. Conditions

Conditions are what you need to make scripts clever. Conditions can evaluate a situation and then take appropriate action. There are a number of condition constructs in the PowerShell language which that we will look at in this chapter.

In the second part, you'll employ conditions to execute PowerShell instructions only if a particular condition is actually met.

Topics Covered:

Creating Conditions

A condition is really just a question that can be answered with yes (true) or no (false). The following PowerShell comparison operators allow you to compare values,

Operator	Coventional	Description	Example	Result
-eq, -ceq, -ieq	=	equals	10 -eq 15	$false
-ne, -cne, -ine	<>	not equal	10 -ne 15	$true
-gt, -cgt, -igt	>	greater than	10 -gt 15	$false
-ge, -cge, -ige	>=	greater than or equal to	10 -ge 15	$false
-lt, -clt, -ilt	<	less than	10 -lt 15	$true
-le, -cle, -ile	<=	less than or equal to	10 -le 15	$true
-contains, -ccontains, -icontains		contains	1,2,3 -contains 1	$true
-notcontains, -cnotcontains, -inotcontains		does not contain	1,2,3 -notcontains 1	$false

Figure 7.1: Comparison operators

Note

PowerShell doesn’t use traditional comparison operators that you may know from other programming languages. In particular, the “=” operator is an assignment operator only in PowerShell, while “>” and “<” operators are used for redirection.

There are three variants of all comparison operators. The basic variant is case-insensitive so it does not distinguish between upper and lower case letters (if you compare text). To explicitly specify whether case should be taken into account, you can use variants that begin with "c" (case-sensitive) or "i" (case-insensitive).

Carrying Out a Comparison

To get familiar with comparison operators, you can play with them in the interactive PowerShell console! First, enter a value, then a comparison operator, and then the second value that you want to compare with the first. When you hit (enter)), PowerShell executes the comparison. The result is always True (condition is met) or False (condition not met).

```
4 -eq 10
False
“secret” -ieq “SECRET”
True
```

As long as you compare only numbers or only strings, comparisons are straight-forward:

```
123 -lt 123.5
True
```

However, you can also compare different data types. However, these results are not always as straight-forward as the previous one:

```
12 -eq “Hello”
False
12 -eq “000012”
True
“12” -eq 12
True
“12” -eq 012
True
“012” -eq 012
False
123 -lt 123.4
True
123 -lt “123.4”
False
123 -lt “123.5”
True
```

Are the results surprising? When you compare different data types, PowerShell will try to convert the data types into one common data type. It will always look at the data type to the left of the comparison operator and then try and convert the value to the right to this data type.

“Reversing” Comparisons

With the logical operator -not you can reverse comparison results. It will expect an expression on the right side that is either true or false. Instead of -not, you can also use “!”:

```
$a = 10
$a -gt 5
True
-not ($a -gt 5)
False

# Shorthand: instead of -not “!” can also be used:
!($a -gt 5)
False
```

Note

You should make good use of parentheses if you're working with logical operators like –not. Logical operators are always interested in the result of a comparison, but not in the comparison itself. That's why the comparison should always be in parentheses.

Combining Comparisons

You can combine several comparisons with logical operators because every comparison returns either True or False. The following conditional statement would evaluate to true only if both comparisons evaluate to true:

```
( ($age -ge 18) -and ($sex -eq "m") )
```

You should put separate comparisons in parentheses because you only want to link the results of these comparisons and certainly not the comparisons themselves

Operator	Description	Left Value	Right Value	Result
-and	Both conditions must be met	True False False True	False True False True	False False False True
-or	At least one of the two conditions must be met	True False False True	False True False True	True True False True
-xor	One or the other condition must be met, but not both	True False False True	True False True False	False False True True
-not	Reverses the result	(not applicable)	True False	False True

Figure 7.2: Logical operators

Comparisons with Arrays and Collections

Up to now, you've only used the comparison operators in Table 7.1 to compare single values. In Chapter 4, you've already become familiar with arrays. How do comparison operators work on arrays? Which element of an array is used in the comparison? The simple answer is all elements!

In this case, comparison operators work pretty much as a filter and return a new array that only contains the elements that matched the comparison.

```
1,2,3,4,3,2,1 -eq 3
3
3
```

If you'd like to see only the elements of an array that don't match the comparison value, you can use -ne (not equal) operator:

```
1,2,3,4,3,2,1 -ne 3
1
2
4
2
```

Verifying Whether an Array Contains a Particular Element

But how would you find out whether an array contains a particular element? As you have seen, -eq provides matching array elements only. -contains and -notcontains. verify whether a certain value exists in an array.

```
# -eq returns only those elements matching the criterion:
1,2,3 -eq 5

# -contains answers the question of whether the sought element is included in the array:
1,2,3 -contains 5
False
1,2,3 -notcontains 5
True
13
```

Where-Object

In the pipeline, the results of a command are handed over to the next one and the Where-Object cmdlet will work like a filter, allowing only those objects to pass the pipeline that meet a certain condition. To make this work, you can specify your condition to Where-Object.

Filtering Results in the Pipeline

The cmdlet Get-Process returns all running processes. If you would like to find out currently running instances of Notepad, you will need to set up the appropriate comparison term. You will first need to know the names of all the properties found in process objects. Here is one way of listing them:

```
Get-Process | Select-Object -first 1 *
__NounName                  : process
Name                        : agrsmsvc
Handles                     : 36
VM                          : 21884928
WS                          : 57344
PM                          : 716800
NPM                         : 1768
Path                        :
```

```
Company                    :
CPU                        :
FileVersion                :
ProductVersion             :
Description                :
Product                    :
Id                         : 1316
PriorityClass              :
HandleCount                : 36
WorkingSet                 : 57344
PagedMemorySize            : 716800
PrivateMemorySize          : 716800
VirtualMemorySize          : 21884928
TotalProcessorTime         :
BasePriority               : 8
ExitCode                   :
HasExited                  :
ExitTime                   :
Handle                     :
MachineName                : .
MainWindowHandle           : 0
MainWindowTitle            :
MainModule                 :
MaxWorkingSet              :
MinWorkingSet              :
Modules                    :
NonpagedSystemMemorySize   : 1768
NonpagedSystemMemorySize64 : 1768
PagedMemorySize64          : 716800
PagedSystemMemorySize      : 24860
PagedSystemMemorySize64    : 24860
PeakPagedMemorySize        : 716800
PeakPagedMemorySize64      : 716800
PeakWorkingSet             : 2387968
PeakWorkingSet64           : 2387968
PeakVirtualMemorySize      : 21884928
PeakVirtualMemorySize64    : 21884928
PriorityBoostEnabled       :
PrivateMemorySize64        : 716800
PrivilegedProcessorTime    :
ProcessName                : agrsmsvc
ProcessorAffinity          :
Responding                 : True
SessionId                  : 0
StartInfo                  : System.Diagnostics.ProcessStartInfo
StartTime                  :
SynchronizingObject        :
Threads                    : {1964, 1000}
UserProcessorTime          :
VirtualMemorySize64        : 21884928
```

```
EnableRaisingEvents          : False
StandardInput                :
StandardOutput               :
StandardError                :
WorkingSet64                 : 57344
Site                         :
Container                    :
```

Putting Together a Condition

As you can see from the previous output, the name of a process can be found in the Name property. If you're just looking for the processes of the Notepad, your condition is: name -eq 'notepad:

```
Get-Process | Where-Object { $_.name -eq 'notepad' }
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  ProcessName
-------- ------  -----    ----- -----   ------       --  -----------
     68       4     1636       8744     62      0,14   7732  notepad
     68       4     1632       8764     62      0,05   7812  notepad
```

Here are two things to note: if the call does not return anything at all, then there are probably no Notepad processes running. Before you make the effort and use Where-Object to filter results, you should make sure the initial cmdlet has no parameter to filter the information you want right away. For example, Get-Process already supports a parameter called -name, which will return only the processes you specify:

```
Get-Process -name notepad
Handles    NPM(K)     PM(K)       WS(K)  VM(M)     CPU(s)     Id    ProcessName
-------  ------     -----       ----- -----     ------     --     -----------
      68         4      1636          8744      62        0,14 7732     notepad
      68         4      1632          8764      62        0,05 7812     notepad
```

The only difference with the latter approach: if no Notepad process is running, Get-Process throws an exception, telling you that there is no such process. If you don't like that, you can always add the parameter -ErrorAction SilentlyContinue, which will work for all cmdlets and hide all error messages.

When you revisit your Where-Object line, you'll see that your condition is specified in curly brackets after the cmdlet. The $_ variable contains the current pipeline object. While sometimes the initial cmdlet is able to do the filtering all by itself (like in the previous example using -name), Where-Object is much more flexible because it can filter on any piece of information found in an object.

You can use the next one-liner to retrieve all processes whose company name begins with "Micro" and output name, description, and company name:

```
Get-Process | Where-Object { $_.company -like 'micro*' } | Select-Object name, description, company
Name                               Description                          Company
----                               -----------                          --------
conime                             Console IME                          Microsoft Corporation
dwm                                Desktopwindow-Manager                Microsoft Corporation
ehmsas                             Media Center Media Status Aggr...    Microsoft Corporation
ehtray                             Media Center Tray Applet             Microsoft Corporation
EXCEL                              Microsoft Office Excel               Microsoft Corporation
explorer                           Windows-Explorer                     Microsoft Corporation
GrooveMonitor                      GrooveMonitor Utility                Microsoft Corporation
ieuser                             Internet Explorer                    Microsoft Corporation
iexplore                           Internet Explorer                    Microsoft Corporation
msnmsgr                            Messenger                            Microsoft Corporation
notepad                            Editor                               Microsoft Corporation
notepad                            Editor                               Microsoft Corporation
sidebar                            Windows-Sidebar                      Microsoft Corporation
taskeng                            Task Scheduler Engine                Microsoft Corporation
WINWORD                            Microsoft Office Word                Microsoft Corporation
wmpnscfg                           Windows Media Player Network S...    Microsoft Corporation
wpcumi                             Windows Parental Control Notif...    Microsoft Corporation
```

Since you will often need conditions in a pipeline, there is an alias for *Where-Object:* "?". So, instead of *Where-Object*, you can also use "?'". However, it does make your code a bit unreadable:

```
# The two following instructions return the same result: all running services
Get-Service | Foreach-Object {$_.Status -eq 'Running' }
Get-Service | ? {$_.Status -eq 'Running' }
```

If-ElseIf-Else

Where-object works great in the pipeline, but it is inappropriate if you want to make longer code segments dependent on meeting a condition. Here, the If..*ElseIf..Else* statement works much better. In the simplest case, the statement will look like this:

```
If (condition) {# If the condition applies, this code will be executed}
```

The condition must be enclosed in parentheses and follow the keyword *If*. If the condition is met, the code in the curly brackets after it will be executed, otherwise, it will not. Try it out:

```
If ($a -gt 10) { "$a is larger than 10" }
```

It's likely, though, that you won't (yet) see a result. The condition was not met, and so the code in the curly brackets wasn't executed. To get an answer, you can make sure that the condition is met:

```
$a = 11
if ($a -gt 10) { "$a is larger than 10" }
11 is larger than 10
```

Now, the comparison is true, and the *If* statement ensures that the code in the curly brackets will return a result. As it is, that clearly shows that the simplest If statement usually doesn't suffice in itself, because you would like to *always* get a result, even when the condition isn't met. You can expand the *If* statement with *Else* to accomplish that:

```
if ($a -gt 10)
{
  "$a is larger than 10"
}
else
{
  "$a is less than or equal to 10"
}
```

Now, the code in the curly brackets after *If* is executed if the condition is met. However, if the preceding condition isn't true, the code in the curly brackets after Else will be executed. If you have several conditions, you may insert as many *ElseIf* blocks between *If* and *Else* as you like:

```
if ($a -gt 10)
{
  "$a is larger than 10"
}
elseif ($a -eq 10)
{
  "$a is exactly 10"
}
else
{
  "$a is less than or equal to 10"
}
```

The *If* statement here will always execute the code in the curly brackets after the condition that is met. The code after *Else* will be executed when none of the preceding conditions are true. What happens if several conditions are true? Then the code after the first applicable condition will be executed and all other applicable conditions will be ignored.

```
if ($a -gt 10)
{
  "$a is larger than 10"
}
elseif ($a -eq 10)
{
  "$a is exactly 10"
}
```

```
else
{
  "$a is smaller than 10"
}
```

Note

The fact is that the If statement doesn't care at all about the condition that you state. All that the *If* statement evaluates is *$true* or *$false*. If condition evaluates *$true*, the code in the curly brackets after it will be executed, otherwise, it will not. Conditions are only a way to return one of the requested values *$true* or *$false*. But the value could come from another function or from a variable:

```
# Returns True from 14:00 on, otherwise False:
function isAfternoon { (get-date).Hour -gt 13 }
isAfternoon
True

# Result of the function determines which code the If statement executes:
if (isAfternoon) { "Time for break!" } else { "It's still early." }
Time for break!
```

This example shows that the condition after If must always be in parentheses, but it can also come from any source as long as it is *$true* or *$false*. In addition, you can also write the *If* statement in a single line. If you'd like to execute more than one command in the curly brackets without having to use new lines, then you should separate the commands with a semi-colon ";".

Switch

If you'd like to test a value against many comparison values, the *If* statement can quickly become unreadable. The *Switch* code is much cleaner:

```
# Test a value against several comparison values (with If statement):
$value = 1
if ($value -eq 1)
{
  " Number 1"
}
elseif ($value -eq 2)
{
  " Number 2"
}
elseif ($value -eq 3)
{
  " Number 3"
}
Number 1
```

```
# Test a value against several comparison values (with Switch statement):
$value = 1
switch ($value)
{
  1  { "Number 1" }
  2  { "Number 2" }
  3  { "Number 3" }
}
Number 1
```

This is how you can use the *Switch statement*: the value to switch on is in the parentheses after the *Switch keyword.* That value is matched with each of the conditions on a case-by-case basis. If a match is found, the action associated with that condition is then performed. You can use the default comparison operator, the *–eq* operator, to verify equality.

Testing Range of Values

The default comparison operator in a switch statement is *-eq*, but you can also compare a value with other comparison statements. You can create your own condition and put it in curly brackets. The condition must then result in either *true* or *false*:

```
$value = 8
switch ($value)
{
  # Instead of a standard value, a code block is used that results in True for numbers smaller than 5:
  {$_ -le 5}  { "Number from 1to 5" }

  # A value is used here; Switch checks whether this value matches $value:
  6  { "Number 6" }

  # Complex conditions areallowed as they are here, where –and is used to combine two comparisons:
  {(($_ -gt 6) -and ($_ -le 10))}  { "Number from 7 to 10" }
}
Number from 7 to 10
```

· The code block *{$_ -le 5}* includes all numbers less than or equal to 5.
· The code block *{(($_ -gt 6)* -and *($_ -le 10))}* combines two conditions and results in true if the number is either larger than 6 or less than-equal to 10. Consequently, you can combine any PowerShell statements in the code block and also use the logical operators listed in Table 7.2.

Here, you can use the initial value stored in *$_* for your conditions, but because *$_* is generally available anywhere in the Switch block, you could just as well have put it to work in the result code:

```
$value = 8
switch ($value)
{
  # The initial value (here it is in $value) is available in the variable $_:
  {$_ -le 5}  { "$_ is a number from 1 to 5" }
  6  { "Number 6" }
  {(($_ -gt 6) -and ($_ -le 10))}  { "$_ is a number from 7 to 10" }
}
8 is a number from 7 to 10
```

Several Applicable Conditions

If more than one condition applies, then *Switch* will work differently from *If*. For *If*, only the first applicable condition was executed. For *Switch*, all applicable conditions are executed:

```
$value = 50
switch ($value)
{
  50  { "the number 50" }
  {$_ -gt 10}  {"larger than 10"}
  {$_ -is [int]}  {"Integer number"}
}
The Number 50
Larger than 10
Integer number
```

Consequently, all applicable conditions will ensure that the following code is executed. So in some circumstances, you may get more than one result.

Tip

Try out that example, but assign 50.0 to $value. In this case, you'll get just two results instead of three. Do you know why? That's right: the third condition is no longer fulfilled because the number in *$value* is no longer an integer number. However, the other two conditions continue to remain fulfilled.

If you'd like to receive only one result, you can add the continue or break statement to the code.

```
$value = 50
switch ($value)
{
  50  { "the number 50"; break }
  {$_ -gt 10}  {"larger than 10"; break}
  {$_ -is [int]}  {"Integer number"; break}
}
The number 50
```

The keyword *break* tells PowerShell to leave the Switch construct. In conditions, *break* and *continue* are interchangeable. In loops, they work differently. While breaks exits a loop immediately, continue would only exit the current iteration.

Using String Comparisons

The previous examples have compared numbers. You could also naturally compare strings since you now know that *Switch* uses only the normal *–eq* comparison operator behind the scenes and that their string comparisons are also permitted.. The following code could be the basic structure of a command evaluation. As such, a different action will be performed, depending on the specified command:

```
$action = "sAVe"
switch ($action)
{
  "save"  { "I save..." }
  "open"  { "I open..." }
  "print"  { "I print..." }
  Default  { "Unknown command" }
}
I save...
```

Case Sensitivity

Since the *–eq* comparison operator doesn't distinguish between lower and upper case, case sensitivity doesn't play a role in comparisons. If you want to distinguish between them, you can use the *–case* option. Working behind the scenes, it will replace the *–eq* comparison operator with *–ceq*, after which case sensitivity will suddenly become crucial:

```
$action = "sAVe"
switch -case ($action)
{
  "save"  { "I save..." }
  "open"  { "I open..." }
  "print"  { "I print..." }
  Default  { "Unknown command" }
}
Unknown command
```

Wildcard Characters

In fact, you can also exchange a standard comparison operator for *–like* and *–match* operators and then carry out wildcard comparisons. Using the *–wildcard* option, you can activate the *-like* operator, which is conversant, among others, with the "*" wildcard character:

```
$text = "IP address: 10.10.10.10"
switch -wildcard ($text)
{
  "IP*"  { "The text begins with IP: $_" }
  "*.*.*.*"  { "The text contains an IP address string pattern: $_" }
  "*dress*"  { "The text contains the string 'dress' in arbitrary locations:
$_" }
}
The text begins with IP: IP address: 10.10.10.10
The text contains an IP address string pattern: IP address: 10.10.10.10
The text contains the string 'dress' in arbitrary locations: IP address: 10.10.10.10
```

Regular Expressions

Simple wildcard characters ca not always be used for recognizing patterns. Regular expressions are much more efficient. But they assume much more basic knowledge, which is why you should take a peek ahead at **Chapter 13**, discussion of regular expression in greater detail.

With the -*regex* option, you can ensure that *Switch* uses the –match comparison operator instead of –*eq*, and thus employs regular expressions. Using regular expressions, you can identify a pattern much more precisely than by using simple wildcard characters. But that's not all!. As in the case with the –*match* operator, you will usually get back the text that matches the pattern in the *$matches* variable. This way, you can even parse information out of the text:

```
$text = "IP address: 10.10.10.10"
switch -regex ($text)
{
  "^IP"  { "The text begins with IP: $($matches[0])" }
  "\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"  { "The text contains an IP address
string pattern: $($matches[0])" }
  "\b.*?dress.*?\b"  { " The text contains the string 'dress' in arbitrary
locations: $($matches[0])" }
}
The text begins with IP: IP address: 10.10.10.10
The text contains an IP address string pattern: IP address: 10.10.10.10
The text contains the string 'dress' in arbitrary locations: IP address: 10.10.10.10
```

Processing Several Values Simultaneously

Until now, you have always passed just one value for evaluation to *Switch*. But *Switch* can also process several values at the same time. To do so, you can pass to Switch the values in an array or a collection. In the following example, *Switch* is passed an array containing five elements. *Switch* will automatically take all the elements, one at a time, from the array and compare each of them, one by one:

```
$array = 1..5
switch ($array)
{
  {$_ % 2} { "$_ is uneven."}
  Default { "$_ is even."}
}
1 is uneven.
2 is even.
3 is uneven.
4 is even.
5 is uneven.
```

There you have it: *Switch* will accept not only single values, but also entire arrays and collections. As such, *Switch* would be an ideal candidate for evaluating results on the PowerShell pipeline because the pipeline character ("|") is used to forward results as arrays or collections from one command to the next.

The next line queries *Get-Process* for all running processes and then pipes the result to a script block *(& {...})*. In the script block, *Switch* will evaluate the result of the pipeline, which is available in *$input*. If the *WS* property of a process is larger than one megabyte, this process is output. *Switch* will then filter all of the processes whose WS property is less than or equal to one megabyte:

```
Get-Process | & { Switch($input) { {$_.WS -gt 1MB} { $_ }}}
```

However, this line is extremely hard to read and seems complicated. You can formulate the condition in a much clearer way by using *Where-Object:*

```
Get-Process | Where-Object { $_.WS -gt 1MB }
```

This variant also works more quickly because *Switch* had to wait until the pipeline has collected the entire results of the preceding command in *$input*. In *Where-Object*, it processes the results of the preceding command precisely when the results are ready. This difference is especially striking for elaborate commands:

```
# Switch returns all files beginning with “a”:
Dir | & { switch($Input) { {$_.name.StartsWith(“a”)} { $_ } }}

# But it doesn’t do so until Dir has retrieved all data, and that can take a long time:
Dir -Recurse | & { switch($Input) { {$_.name.StartsWith(“a”)} { $_ } }}

# Where-Object processes the incoming results immediately:
Dir -recurse | Where-Object { $_.name.StartsWith(“a”) }

# The alias of Where-Object (“?”) works exactly the same way:
Dir -recurse | ? { $_.name.StartsWith(“a”) }
```

Summary

Intelligent decisions are based on conditions, which in their simplest form can be reduced to plain *Yes* or *No* answers. Using the comparison operators listed in Table 7.1, you can formulate such conditions and even combine these with the logical operators listed in Table 7.2 to form complex queries.

The simple *Yes/No* answers of your conditions will determine whether particular PowerShell instructions can carried out or not. In their simplest form, you can use the Where-Object cmdlet in the pipeline. It functions there like a filter, allowing only those results through the pipeline that correspond to your condition.

If you would like more control, or would like to execute larger code segments independently of conditions, you can use the *If* statement, which evaluates as many different conditions as you wish and, depending on the result, will then execute the allocated code. This is the typical “If-Then” scenario: if certain conditions are met, *then* certain code segments will be executed.

An alternative to the *If* statement is the *Switch* statement. Using it, you can compare a fixed initial value with various possibilities. *Switch* is the right choice when you want to check a particular variable against many different possible values.

Chapter 8. Loops

Loops repeat PowerShell code and are the heart of automation. In this chapter, you will learn the PowerShell loop constructs.

Topics Covered:

ForEach-Object

Many PowerShell cmdlets return more than one result object. You can use a Pipeline loop: *foreach-object* to process them all one after another.. In fact, you can easily use this loop to repeat the code multiple times. The next line will launch 10 instances of the Notepad editor:

```
1..10 | Foreach-Object { notepad }
```

Foreach-Object is simply a cmdlet, and the script block following it really is an argument assigned to *Foreach-Object:*

```
1..10 | Foreach-Object -process { notepad }
```

Inside of the script block, you can execute any code. You can also execute multiple lines of code. You can use a semicolon to separate statements from each other in one line:

```
1..10 | Foreach-Object { notepad; "Launching Notepad!" }
```

In PowerShell editor, you can use multiple lines:

```
1..10 | Foreach-Object { notepad "Launching Notepad!" }
```

The element processed by the script block is available in the special variable $_:

```
1..10 | Foreach-Object { "Executing $_. Time" }
```

Most of the time, you will not feed numbers into Foreach-Object, but instead the results of another cmdlet. Have a look:

```
Get-Process | Foreach-Object { 'Process {0} consumes {1} seconds CPU time' -f $_.Name, $_.CPU }
```

Invoking Methods

Because *ForEach-Object* will give you access to each object in a pipeline, you can invoke methods of these objects. In Chapter 7, you learned how to take advantage of this to close all instances of the Notepad. This will give you much more control. You could use *Stop-Process* to stop a process. But if you want to close programs gracefully, you should provide the user with the opportunity to save unsaved work by also invoking the method *CloseMainWindow()*. The next line closes all instances of Notepad windows. If there is unsaved data, a dialog appears asking the user to save it first:

```
Get-Process notepad | ForEach-Object { $_.CloseMainWindow() }
```

You can also solve more advanced problems. If you want to close only those instances of Notepad that were running for more than 10 minutes, you can take advantage of the property StartTime. All you needed to do is calculate the cut-off date using *New-Timespan.* Let's first get a listing that tells you how many minutes an instance of Notepad has been running:

```
Get-Process notepad | ForEach-Object {
  $info = $_ | Select-Object Name, StartTime, CPU, Minutes
  $info.Minutes = New-Timespan $_.StartTime | Select-Object -expandproperty TotalMinutes
  $info
}
```

Check out a little trick. In the above code, the script block creates a copy of the incoming object using *Select-Object*, which selects the columns you want to view. We specified an additional property called Minutes to display the running minutes, which are not part of the original object. *Select-Object* will happily add that new property to the object. Next, we can fill in the information into the *Minutes* property. This is done using *New-Timespan*, which calculates the time difference between now and the time found in *StartTime*. Don't forget to output the *$info* object at the end or the script block will have no result.

To kill only those instances of Notepad that were running for more than 10 minutes, you will need a condition:

```
Get-Process Notepad | Foreach-Object {
  $cutoff = ( (Get-Date) - (New-Timespan -minutes 10) )
  if ($_.StartTime -lt $cutoff) { $_ }
}
```

This code would only return Notepad processes running for more than 10 minutes and you could pipe the result into *Stop-Process* to kill those.

What you see here is a *Foreach-Object* loop with an If condition. This is exactly what *Where-Object* does so if you need loops with conditions to filter out unwanted objects, you can simplify:

```
Get-Process Notepad | Where-Object {
  $cutoff = ( (Get-Date) - (New-Timespan -minutes 10) )
  $_.StartTime -lt $cutoff
}
```

Foreach

There is another looping construct called *Foreach*. Don't confuse this with the *Foreach* alias, which represents Foreach-Object. So, if you see a Foreach statement inside a pipeline, this really is a *Foreach-Object* cmdlet. The true Foreach loop is never used inside the pipeline. Instead, it can only live inside a code block.

While *Foreach-Object* obtains its entries from the pipeline, the *Foreach* statement iterates over a collection of objects:

```
# ForEach-Object lists each element in a pipeline:
Dir C:\ | ForEach-Object { $_.name }

# Foreach loop lists each element in a colection:
foreach ($element in Dir C:\) { $element.name }
```

The true *Foreach* statement does not use the pipeline architecture. This is the most important difference because it has very practical consequences. The pipeline has a very low memory footprint because there is always only one object travelling the pipeline. In addition, the pipeline processes objects in real time. That's why it is safe to process even large sets of objects. The following line iterates through all files and folders on drive c:\. Note how results are returned immediately:

```
Dir C:\ -recurse -erroraction SilentlyContinue | ForEach-Object { $_.FullName }
```

If you tried the same with foreach, the first thing you will notice is that there is no output for a long time. Foreach does not work in real time. So, it first collects all results before it starts to iterate. If you tried to enumerate all files and folders on your drive c:\, chances are that your system runs out of memory before it has a chance to process the results. You must be careful with the following statement:

```
# careful!
foreach ($element in Dir C:\ -recurse -erroraction SilentlyContinue) { $element.FullName }
```

On the other hand, foreach is much faster than foreach-object because the pipeline has a significant overhead. It is up to you to decide whether you need memory efficient real-time processing or fast overall performance:

```
Measure-Command { 1..10000 | Foreach-Object { $_ } } | Select-Object -expandproperty TotalSeconds
0,9279656
Measure-Command { foreach ($element in (1..10000)) { $element } } | Select-Object -expandproperty TotalSeconds
0,0391117
```

Do and While

Do and *While* generate endless loops. Endless loops are a good idea if you don't know exactly how many times the loop should iterate. You must set additional abort conditions to prevent an endless loop to really run endlessly. The loop will end when the conditions are met.

Continuation and Abort Conditions

A typical example of an endless loop is a user query that you want to iterate until the user gives a valid answer. How long that lasts and how often the query will iterate depends on the user and his ability to grasp what you want.

```
do {
  $Input = Read-Host "Your homepage"
} while (!($Input -like "www.*.*"))
```

This loop asks the user for his home page Web address. *While* is the criteria that has to be met at the end of the loop so that the loop can be iterated once again. In the example, *-like* is used to verify whether the input matches the www.*.* pattern. While that's only an approximate verification, it usually suffices. You could also use regular expressions to refine your verification. Both procedures will be explained in detail in *Chapter 13*.

This loop is supposed to re-iterate only if the input is false. That's why "!" is used to simply invert the result of the condition. The loop will then be iterated until the input does *not* match a Web address.

In this type of endless loop, verification of the loop criteria doesn't take place until the end. The loop will go through its iteration at least once because you have to query the user at least once before you can check the criteria.

There are also cases in which the criteria needs to be verified at the beginning and not at the end of the loop. An example would be a text file that you want to read one line at a time. The file could be empty and the loop should check before its first iteration whether there's anything at all to read. To accomplish this, just put the *While* statement and its criteria at the beginning of the loop (and leave out *Do*, which is no longer of any use):

```
# Open a file for reading:
$file = [system.io.file]::OpenText("C:\autoexec.bat")

# Continue loop until the end of the file has been reached:
while (!($file.EndOfStream)) {

  # Read and output current line from the file:
  $file.ReadLine()
}

# Close file again:
$file.close
```

Using Variables as Continuation Criteria

The truth is that the continuation criteria after *While* works like a simple switch. If the expression is *$true*, then the loop will be iterated; if it is $false, then it won't. Conditions are therefore not mandatory, but simply provide the required *$true* or *$false*. You could just as well have presented the loop with a variable instead of a comparison operation, as long as the variable contained *$true* or *$false.*

```
do {
  $Input = Read-Host "Your Homepage"
  if ($Input -like "www.*.*") {
    # Input correct, no further query:
    $furtherquery = $false
  } else {
    # Input incorrect, give explanation and query again:
    Write-Host -Fore "Red" "Please give a valid web address."
    $furtherquery = $true
  }
} while ($furtherquery)
Your Homepage: hjkh
Please give a valid web address.
Your Homepage: www.powershell.com
```

Endless Loops without Continuation Criteria

You can also omit continuation criteria and instead simply use the fixed value *$true* after *While*. The loop will then become a genuinely endless loop, which will never stop on its own. Of course, that makes sense only if you exit the loop in some other way. The *break* statement can be used for this:

```
while ($true) {
  $Input = Read-Host "Your homepage"
  if ($Input -like "www.*.*") {
```

```
# Input correct, no further query:
    break
  } else {
    # Input incorrect, give explanation and ask again:
    Write-Host -Fore "Red" "Please give a valid web address."
  }
}
Your homepage: hjkh
Please give a valid web address.
Your homepage: www.powershell.com
```

For

You can use the *For* loop if you know exactly how often you want to iterate a particular code segment. *For* loops are counting loops. You can specify the number at which the loop begins and at which number it will end to define the number of iterations, as well as which increments will be used for counting. The following loop will output a sound at various 100ms frequencies (provided you have a soundcard and the speaker is turned on):

```
# Output frequencies from 1000Hz to 4000Hz in 300Hz increments
for ($frequency=1000; $frequency -le 4000; $frequency +=300) {
  [System.Console]::Beep($frequency,100)
}
```

For Loops: Just Special Types of the While Loop

If you take a closer look at the *For* loop, you'll quickly notice that it is actually only a specialized form of the *While* loop. The *For* loop, in contrast to the *While* loop, evaluates not only one, but three expressions:

· Initialization: The first expression is evaluated when the loop begins.
· Continuation criteria: The second expression is evaluated before every iteration. It basically corresponds to the continuation criteria of the *While* loop. If this expression is *$true*, the loop will iterate.
· Increment: The third expression is likewise re-evaluated with every looping, but it is not responsible for iterating. Be careful as this expression cannot generate output.

These three expressions can be used to initialize a control variable, to verify whether a final value is achieved, and to change a control variable with a particular increment at every iteration of the loop. Of course, it is entirely up to you whether you want to use the For loop solely for this purpose.

A *For* loop can become a *While* loop if you ignore the first and the second expression and only use the second expression, the continuation criteria:

```
# First expression: simple While loop:
$i = 0
while ($i -lt 5) {
  $i++
  $i
}
1
2
3
4
5

# Second expression: the For loop behaves like the While loop:
$i = 0
for (;$i -lt 5;) {
  $i++
 $i
}
1
2
3
4
5
```

Unusual Uses for the For Loop

Of course in this case, it might have been preferable to use the *While* loop right from the start. It certainly makes more sense not to ignore the other two expressions of the *For* loop, but to use them for other purposes. The first expression of the *For* loop can be generally used for initialization tasks. The third expression sets the increment of a control variable, as well as performs different tasks in the loop. In fact, you can also use it in the user query example we just reviewed:

```
for ($Input=""; !($Input -like "www.*.*"); $Input = Read-Host "Your homepage") {
  Write-Host -fore "Red" " Please give a valid web address."
}
```

In the first expression, the *$input* variable is set to an empty string. The second expression checks whether a valid Web address is in *$input*. If it is, it will use "!" to invert the result so that it is $true if an invalid Web address is in *$input*. In this case, the loop is iterated. In the third expression, the user is queried for a Web address. Nothing more needs to be in the loop. In the example, an explanatory text is output.

In addition, the line-by-line reading of a text file can be implemented by a *For* loop with less code:

```
for ($file = [system.io.file]::OpenText("C:\autoexec.bat"); !($file.EndOfStream); `
$line = $file.ReadLine())
{
  # Output read line:
  $line
}
$file.close()
REM Dummy file for NTVDM
```

In this example, the first expression of the loop opened the file so it could be read. In the second expression, a check is made whether the end of the file has been reached. The "!" operator inverts the result again. It will return *$true* if the end of the file hasn't been reached yet so that the loop will iterate in this case. The third expression reads a line from the file. The read line is then output in the loop.

Note

The third expression of the *For* loop is executed before every loop cycle. In the example, the current line from the text file is read. This third expression is always executed invisibly, which means you can't use it to output any text. So, the contents of the line are output within the loop.

Switch

Switch is not only a condition, but also functions like a loop. That makes *Switch* one of the most powerful statements in PowerShell. *Switch* works almost exactly like the *Foreach* loop. Moreover, it can evaluate conditions. For a quick demonstration, take a look at the following simple *Foreach* loop:

```
$array = 1..5
foreach ($element in $array)
{
  "Current element: $element"
}
Current element: 1
Current element: 2
Current element: 3
Current element: 4
Current element: 5
```

If you use switch, this loop would look like this:

```
$array = 1..5
switch ($array)
{
  Default { "Current element: $_" }
}
Current element: 1
Current element: 2
Current element: 3
Current element: 4
Current element: 5
```

The control variable that returns the current element of the array for every loop cycle cannot be named for *Switch*, as it can for *Foreach*, but is always called $_. The external part of the loop functions in exactly the same way. Inside the loop, there's an additional difference: while *Foreach* always executes the same code every time the loop cycles, *Switch* can utilize conditions to execute optionally different code for every loop. In the simplest case, the *Switch* loop contains only the *default* statement. The code that is to be executed follows it in curly brackets.

That means *Foreach* is the right choice if you want to execute exactly the same statements for every loop cycle. On the other hand, if you'd like to process each element of an array according to its contents, it would be preferable to use *Switch*:

```
$array = 1..5
switch ($array)
{
  1  { "The number 1" }
  {$_ -lt 3}  { "$_ is less than 3" }
  {$_ % 2}  { "$_ is odd" }
  Default { "$_ is even" }
}
The number 1
1 is less than 3
1 is odd
2 is less than 3
3 is odd
4 is even
5 is odd
```

If you're wondering why *Switch* returned this result, take a look at **Chapter 7** where you'll find an explanation of how *Switch* evaluates conditions. What's important here is the other, loop-like aspect of *Switch.*

Exiting Loops Early

You can exit all loops by using the *Break* statement, which will give you the additional option of defining additional stop criteria in the loop. The following is a little example of how you can ask for a password and then use *Break* to exit the loop as soon as the password "secret" is entered.

```
while ($true)
{
  $password = Read-Host "Enter password"
  if ($password -eq "secret") {break}
}
```

Continue: Skipping Loop Cycles

The *Continue* statement aborts the current loop cycle, but does continue the loop. The next example shows you how to abort processing folders and only focus on files returned by Dir:

```
foreach ($entry in Dir $env:windir)
{
  # If the current element matches the desired type, continue immediately with the next element:
  if (!($entry -is [System.IO.FileInfo])) { continue }

  "File {0} is {1} bytes large." -f $entry.name, $entry.length
}
```

Of course, you can also use a condition to filter out sub-folders:

```
foreach ($entry in Dir $env:windir)
{
  if ($entry -is [System.IO.FileInfo]) {
    "File {0} is {1} bytes large." -f $entry.name, $entry.length
  }
}
```

This also works in a pipeline using a Where-Object condition:

```
Dir $env:windir | Where-Object { $_ -is [System.IO.FileInfo] }
```

Nested Loops and Labels

Loops may be nested within each other. However, if you do nest loops, how do *Break* and *Continue* work? They will always affect the inner loop, which is the loop that they were called from. However, you can label loops and then submit the label to continue or break if you want to exit or skip outer loops, too.

The next example nests two *Foreach* loops. The first (outer) loop cycles through a field with three WMI class names. The second (inner) loop runs through all instances of the respective WMI class. This allows you to output all instances of all three WMI classes. The inner loop checks whether the name of the current instance begins with "a"; if not, the inner loop will then invoke *Continue* skip all instances not beginning with "a." The result is a list of all services, user accounts, and running processes that begin with "a":

```
foreach ($wmiclass in "Win32_Service","Win32_UserAccount","Win32_Process")
{
  foreach ($instance in Get-WmiObject $wmiclass) {
    if (!(($instance.name.toLower()).StartsWith("a"))) {continue}
    "{0}: {1}" -f $instance.__CLASS, $instance.name
  }
}
Win32_Service: AeLookupSvc
Win32_Service: AgereModemAudio
Win32_Service: ALG
Win32_Service: Appinfo
Win32_Service: AppMgmt
Win32_Service: Ati External Event Utility
Win32_Service: AudioEndpointBuilder
Win32_Service: Audiosrv
Win32_Service: Automatic LiveUpdate - Scheduler
Win32_UserAccount: Administrator
```

```
Win32_Process: Ati2evxx.exe
Win32_Process: audiodg.exe
Win32_Process: Ati2evxx.exe
Win32_Process: AppSvc32.exe
Win32_Process: agrsmsvc.exe
Win32_Process: ATSwpNav.exe
```

As expected, the *Continue* statement in the inner loop has had an effect on the inner loop where the statement was contained. But how would you change the code if you'd like to see only the first element of all services, user accounts, and processes that begins with "a"? Actually, you would do almost the exact same thing, except now Continue would need to have an effect on the outer loop. Once an element was found that begins with "a," the outer loop would continue with the next WMI class:

```
:WMIClasses foreach ($wmiclass in "Win32_Service","Win32_UserAccount","Win32_Process") {
  :ExamineClasses foreach ($instance in Get-WmiObject $wmiclass) {
    if (($instance.name.toLower()).StartsWith("a")) {
      "{0}: {1}" -f $instance.__CLASS, $instance.name
      continue WMIClasses
    }
  }
}
Win32_Service: AeLookupSvc
Win32_UserAccount: Administrator
Win32_Process: Ati2evxx.exe
```

Summary

The cmdlet *ForEach-Object* will give you the option of processing single objects of the PowerShell pipeline, such as to output the data contained in object properties as text or to invoke methods of the object. *Foreach* is a similar type of loop whose contents do not come from the pipeline, but from an array or a collection.

In addition, there are endless loops that iterate a code block until a particular condition is met. The simplest type is *While*, which checks its continuation criteria at the beginning of the loop. If you want to do the checking at the end of the loop, choose *Do…While.* The *For* loop is an extended *While* loop, because it can count loop cycles and automatically terminate the loop after a designated number of iterations.

This means that *For* is best suited for loops which need to be counted or must complete a set number of iterations. On the other hand, *Do...While* and *While* are designed for loops that have to be iterated as long as the respective situation and running time conditions require it.

Finally, *Switch* is a combined Foreach loop with integrated conditions so that you can immediately implement different actions independently of the read element. Moreover, *Switch* can step through the contents of text files line-by-line and evaluate even log files of substantial size.

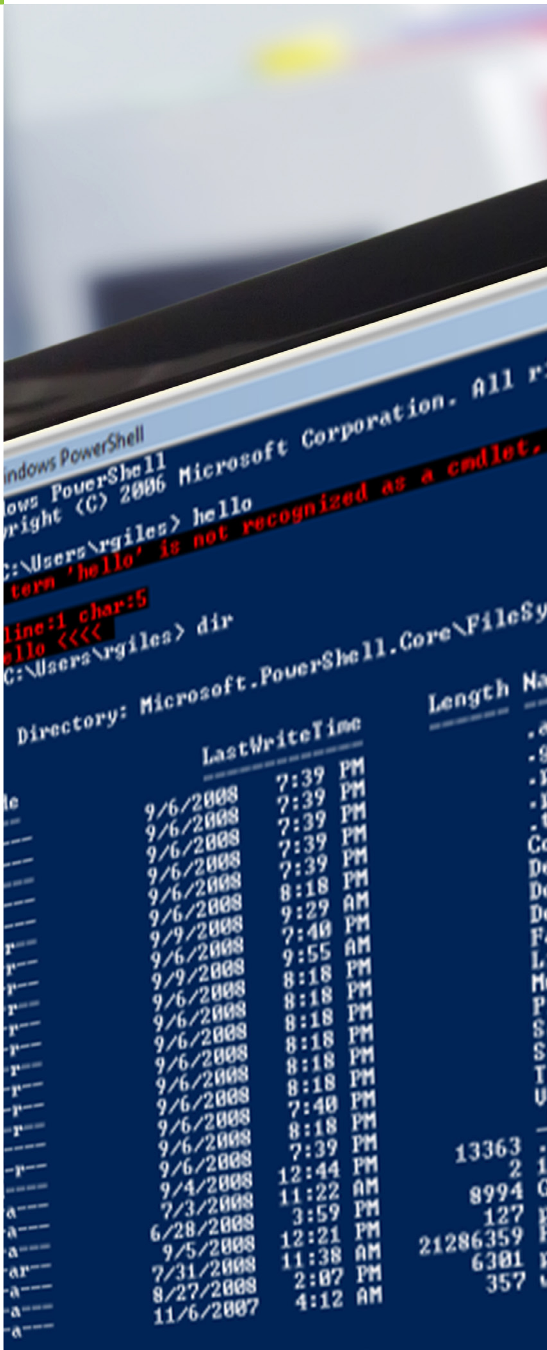
All loops can exit ahead of schedule with the help of Break and skip the current loop cycle with the help of *Continue*. In the case of nested loops, you can assign an unambiguous name to the loops and then use this name to apply *Break* or *Continue* to nested loops.

Chapter 9. Functions

Functions work pretty much like macros. As such, you can attach a script block to a name to create your own new commands.

Functions provide the interface between your code and the user. They can define parameters, parameter types, and even provide help, much like cmdlets.

In this chapter, you will learn how to create your own functions.



Topics Covered:

· Creating New Functions
· Defining Function Parameters
· Adding Mandatory Parameters
· Adding Switch Parameters
· Adding Help to your Functions
· Creating Pipeline-Aware Functions
· Playing With Prompt Functions

Creating New Functions

The most simplistic function consists of a name and a script block. Whenever you call that name, the script block executes. Let's create a function that reads installed software from your registry.

```
function Get-InstalledSoftware {

}
```

Once you enter this code in your script editor and run it dot-sourced, PowerShell learned a new command called *Get-InstalledSoftware*. If you saved your code in a file called *c:\somescript.ps*1, you will need to run it like this:

```
. 'c:\somescript.ps1'
```

If you don't want to use a script, you can also enter a function definition directly into your interactive PowerShell console like this:

```
function Get-InstalledSoftware {  }
```

However, defining functions in a script is a better approach because you won't want to enter your functions manually all the time. Running a script to define the functions is much more practical. You may want to enable script execution if you are unable to run a script because of your current *ExecutionPolicy* settings:

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

Once you defined your function, you can even use code completion. If you enter "Get-Ins" and then press *TAB*, PowerShell will complete your function name. Of course, the new command *Get-InstalledSoftware* won't do anything yet. The script block you attached to your function name was empty. You can add whatever code you want to run to make your function do something useful. Here is the beef to your function that makes it report installed software:

```
function Get-InstalledSoftware
  $path = 'Registry::HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\*'
  Get-ItemProperty -path $path |
    Where-Object { $_.DisplayName -ne $null } |
    Select-Object DisplayName, DisplayVersion, UninstallString |
    Sort-Object DisplayName
}
```

When you run it, it will return a sorted list of all the installed software packages, their version, and their uninstall information:

```
PS > Get-InstalledSoftware

DisplayName                         DisplayVersion             UninstallString
-----------                         --------------             ---------------
64 Bit HP CIO Components Installer  8.2.1                      MsiExec.exe /I{5737101A-27C4-40...
Apple Mobile Device Support         3.3.0.69                   MsiExec.exe /I{963BFE7E-C350-43...
Bonjour                             2.0.4.0                    MsiExec.exe /X{E4F5E48E-7155-4C...
(...)
```

As always, information may be clipped. You can pipe the results to any of the formatting cmdlets to change because the information returned by your function will behave just like information returned from any cmdlet.

```
PS > function test { "One" }
PS > test
One
PS > function test { "Zero", "One", "Two", "Three" }
PS > test
Zero
One
Two
Three
PS > $result = test
PS > $result[0]
Zero
PS > $result[1,2]
One
Two
PS > $result[-1]
Three
```

Defining Function Parameters

Some functions, such as *Get-InstalledSoftware* in the previous example, will work without additional information from the user. From working with cmdlets, you already know how clever it can be to provide detailed information so the command can return exactly what you want. So, let's try adding some parameters to our function.

Adding parameters is very simple. You can either add them in parenthesis right behind the function name or move the list of parameters inside your function and label this part param. Both definitions define the same function:

```
function Speak-Text ($text) {
  (New-Object -com SAPI.SPVoice).Speak($text) | Out-Null
}

function Speak-Text {
 param ($text)

  (New-Object -com SAPI.SPVoice).Speak($text) | Out-Null
}
```

Your new command *Speak-Text* converts (English) text to spoken language. It accesses an internal Text-to-Speech-API, so you can now try this:

```
Speak-Text 'Hello, I am hungry!'
```

Since the function *Speak-Text* now supports a parameter, it is easy to submit additional information to the function code. PowerShell will take care of parameter parsing, and the same rules apply that you already know from cmdlets. You can submit arguments as named parameters, as abbreviated named parameters, and as positional parameters:

```
Speak-Text 'This is positional'
Speak-Text -text 'This is named'
Speak-Text -t 'This is abbreviated named'
```

To submit more than one parameter, you can add more parameters as comma-separated list. Let's add some parameters to *Get-InstalledSoftware* to make it more useful. Here, we add parameters to select the product and when it was installed:

```
function Get-InstalledSoftware {
  param(
    $name = '*',

    $days = 2000
  )

  $cutoff = (Get-Date) - (New-TimeSpan -days $days)
  $cutoffstring = Get-Date -date $cutoff -format 'yyyyMMdd'

  $path = 'Registry::HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\*'
  $column_days = @{
    Name='Days'
    Expression={
          if  ($_.InstallDate) {
            (New-TimeSpan ([DateTime]::ParseExact($_.InstallDate, 'yyyyMMdd', $null))).Days
            } else { 'n/a' }
        }
    }
  Get-ItemProperty -path $path |
    Where-Object { $_.DisplayName -ne $null } |
    Where-Object { $_.DisplayName -like $name } |
    Where-Object { $_.InstallDate -gt $cutoffstring } |
    Select-Object DisplayName, $column_Days, DisplayVersion |
    Sort-Object DisplayName
}
```

Now, *Get-InstalledSoftware* supports two optional parameters called *-Name* and *-Days*. You do not have to submit them since they are optional. If you don't, they are set to their default values. So when you run *Get-InstalledSoftware*, you will get all software installed within the past 2,000 days. If you want to only find software with "Microsoft" in its name that was installed within the past 180 days, you can submit parameters:

```
PS > Get-InstalledSoftware -name *Microsoft* -days 180 | Format-Table -AutoSize

DisplayName                                         Days DisplayVersion
-----------                                        ---- --------------
Microsoft .NET Framework 4 Client Profile            38 4.0.30319
Microsoft Antimalware                               119 3.0.8107.0
Microsoft Antimalware Service DE-DE Language Pack  119 3.0.8107.0
Microsoft Security Client                           119 2.0.0657.0
Microsoft Security Client DE-DE Language Pack      119 2.0.0657.0
Microsoft Security Essentials                       119 2.0.657.0
Microsoft SQL Server Compact 3.5 SP2 x64 ENU         33 3.5.8080.0
```

Adding Mandatory Parameters

Let’s assume you want to create a function that converts dollars to Euros . Here is a simple version:

```
function ConvertTo-Euro {
  param(
    $dollar,

    $rate=1.37
  )

  $dollar * $rate
}
```

And here is how you run your new command:

```
PS > ConvertTo-Euro -dollar 200
274
```

Since *-rate* is an optional parameter with a default value, there is no need for you to submit it unless you want to override the default value:

```
PS > ConvertTo-Euro -dollar 200 -rate 1.21
274
```

So, what happens when the user does not submit any parameter since *-dollar* is optional as well? Well, since you did not submit anything, you get back nothing.

This function can only make sense if there was some information passed to *$dollar*, which is why this parameter needs to be mandatory. Here is how you declare it mandatory:

```
function ConvertTo-Euro {
  param(
    [Parameter(Mandatory=$true)]
    $dollar,

    $rate=1.37
  )

  $dollar * $rate
}
```

This works because PowerShell will ask for it when you do not submit the -dollar parameter:

```
cmdlet ConvertTo-Euro at command pipeline position 1
Supply values for the following parameters:
dollar: 100
100100100100100100100
```

However, the result looks strange because when you enter information via a prompt, PowerShell will treat it as string (text) information, and when you multiply texts, they are repeated. So whenever you declare a parameter as mandatory, you are taking the chance that the user will omit it and gets prompted for it. So, you always need to make sure that you declare the target type you are expecting:

```
function ConvertTo-Euro {
  param(
    [Parameter(Mandatory=$true)]
    [Double]
    $dollar,

    $rate=1.37
  )

  $dollar * $rate
}
```

Now, the function performs as expected:

```
PS > ConvertTo-Euro -rate 6.7

cmdlet ConvertTo-Euro at command pipeline position 1
Supply values for the following parameters:
dollar: 100
670
```

Adding Switch Parameters

There is one parameter type that is special: switch parameters do not take arguments. They are either present or not. You can assign the type [Switch] to that parameter to add switch parameters to your function. If you wanted to provide a way for your users to distinguish between raw and pretty output, your currency converter could implement a switch parameter called -Pretty. When present, the output would come as a nice text line, and when it is not present, it would be the raw numeric value:

```
function ConvertTo-Euro {
  param(
    [Parameter(Mandatory=$true)]
    [Double]
    $dollar,

    $rate=1.37,

    [switch]
    $pretty
  )

  $result = $dollar * $rate
  if ($pretty) {
    '${0:0.00} equals EUR{1:0.00} at a rate of {2:0:0.00}' -f
      $dollar, $result, $rate
  } else {
    $result
  }
}
```

Now, it is up to your user to decide which output to choose:

```
PS > ConvertTo-Euro -dollar 200 -rate 1.28
256
PS > ConvertTo-Euro -dollar 200 -rate 1.28 -pretty
$200,00 equals EUR256,00 at a rate of 1.28
```

Adding Help to your Functions

Get-Help returns Help information for all of your cmdlets. It can also return Help information for your self-defined functions. All you will need to do is add the Help text. To do that, add a specially formatted comment block right before the function or at the beginning or end of the function script block:

```
<#
.SYNOPSIS
   Converts Dollar to Euro
.DESCRIPTION
   Takes dollars and calculates the value in Euro by applying an exchange rate
.PARAMETER dollar
   the dollar amount. This parameter is mandatory.
.PARAMETER rate
   the exchange rate. The default value is set to 1.37.
.EXAMPLE
   ConvertTo-Euro 100
   converts 100 dollars using the default exchange rate and positional parameters
.EXAMPLE
   ConvertTo-Euro 100 -rate 2.3
   converts 100 dollars using a custom exchange rate
#>
```

```
function ConvertTo-Euro {
  param(
    [Parameter(Mandatory=$true)]
    [Double]
    $dollar,

    $rate=1.37,

    [switch]
    $pretty
  )

  $result = $dollar * $rate

  if ($pretty) {
    '${0:0.00} equals EUR{1:0.00} at a rate of {2:0:0.00}' -f
      $dollar, $result, $rate
  } else {
    $result
  }
}
```

Note that the comment-based Help block may not be separated by more than one blank line if you place it above the function. If you did everything right, you will now be able to get the same rich help like with cmdlets after running the code:

```
PS > ConvertTo-Euro -?

NAME
    ConvertTo-Euro

SYNOPSIS
    Converts Dollar to Euro


SYNTAX
    ConvertTo-Euro [-dollar] <Double> [[-rate] <Object>] [-pretty] [<CommonParameters>]


DESCRIPTION
    Takes dollars and calculates the value in Euro by applying an exchange rate


RELATED LINKS

REMARKS
    To see the examples, type: "get-help ConvertTo-Euro -examples".
    for more information, type: "get-help ConvertTo-Euro -detailed".
    for technical information, type: "get-help ConvertTo-Euro -full".



PS > Get-Help -name ConvertTo-Euro -Examples
```

```
NAME
    ConvertTo-Euro

SYNOPSIS
    Converts Dollar to Euro

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

    C:\PS>ConvertTo-Euro 100


    converts 100 dollars using the default exchange rate and positional parameters




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

    C:\PS>ConvertTo-Euro 100 -rate 2.3


    converts 100 dollars using a custom exchange rate

PS > Get-Help -name ConvertTo-Euro -Parameter *

-dollar <Double>
    the dollar amount. This parameter is mandatory.

    Required?                    true
    Position?                    1
    Default value
    Accept pipeline input?       false
    Accept wildcard characters?


-rate <Object>
    the exchange rate. The default value is set to 1.37.

    Required?                    false
    Position?                    2
    Default value
    Accept pipeline input?       false
    Accept wildcard characters?


-pretty [<SwitchParameter>]

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

Creating Pipeline-Aware Functions

Your function is not yet pipeline aware/ So, it will simply ignore the results delivered by the upstream cmdlet if you call it within a pipeline statement:

```
1..10 | ConvertTo-Euro
```

Instead, you will receive exceptions complaining about PowerShell not being able to “bind” the input object. That’s because PowerShell cannot know which parameter is supposed to receive the incoming pipeline values. If you want your function to be pipeline aware, you can fix it by choosing the parameter that is to receive the pipeline input. Here is the enhanced *param* block:

```
function ConvertTo-Euro {
  param(
    [Parameter(Mandatory=$true, ValueFromPipeline=$true)]
    [Double]
    $dollar,

    $rate=1.37,

    [switch]
    $pretty
  )
...
```

By adding *ValueFromPipeline=$true*, you are telling PowerShell that the parameter -dollar is to receive incoming pipeline input. When you rerun the script and then try the pipeline again, there are no more exceptions. Your function will only process the last incoming result, though:

```
PS > 1..10 | ConvertTo-Euro
13,7
```

This is because functions will by default execute all code at the end of a pipeline. If you want the code to process each incoming pipeline data, you must assign the code manually to a process script block or rename your function into a filter (by exchanging the keyword *function by filter*). Filters will by default execute all code in a *process* block.

Here is how you move the code into a process block to make a function process all incoming pipeline values:

```
<#
.SYNOPSIS
   Converts Dollar to Euro
.DESCRIPTION
   Takes dollars and calculates the value in Euro by applying an exchange rate
.PARAMETER dollar
   the dollar amount. This parameter is mandatory.
.PARAMETER rate
   the exchange rate. The default value is set to 1.37.
```

```
.EXAMPLE
   ConvertTo-Euro 100
   converts 100 dollars using the default exchange rate and positional parameters
.EXAMPLE
   ConvertTo-Euro 100 -rate 2.3
   converts 100 dollars using a custom exchange rate
#>
function ConvertTo-Euro {
    param(
    [Parameter(Mandatory=$true, ValueFromPipeline=$true)]
    [Double]
    $dollar,

    $rate = 1.37,

    [switch]
    $pretty
    )
  begin {"starting..."}

  process {
        $result = $dollar * $rate

        if ($pretty) {
            '${0:0.00} equals EUR{1:0.00} at a rate of {2:0:0.00}' -f
            $dollar, $result, $rate
        } else {
            $result
        }
    }

  end { "Done!" }

}
```

As you can see, your function code is now assigned to one of three special script blocks: begin, process, and end. Once you add one of these blocks, no code will exist outside of any one of these three blocks anymore.

Playing With Prompt Functions

PowerShell already contains some pre-defined functions. You can enumerate the special drive function if you would like to see all available functions:

```
Dir function:
```

Many of these pre-defined functions perform important tasks in PowerShell. The most important place for customization is the function *prompt*, which is executed automatically once a command is done. It is responsible for displaying the PowerShell prompt. You can change your PowerShell prompt by overriding the function prompt. This will get you a colored prompt:

```
function prompt
{
    Write-Host ("PS " + $(get-location) +">") -nonewline -foregroundcolor Magenta
    " "
}
```

You can also insert information into the console screen buffer. This only works with true consoles so you cannot use this type of prompt in non-console editors, such as PowerShell ISE.

```
function prompt
{
    Write-Host ("PS " + $(get-location) +">") -nonewline -foregroundcolor Green
    " "
    $winHeight = $Host.ui.rawui.WindowSize.Height
    $curPos = $Host.ui.rawui.CursorPosition
    $newPos = $curPos
    $newPos.X = 0
    $newPos.Y-=$winHeight
    $newPos.Y = [Math]::Max(0, $newPos.Y+1)
    $Host.ui.rawui.CursorPosition = $newPos
    Write-Host ("{0:D} {0:T}" -f (Get-Date)) -foregroundcolor Yellow
    $Host.ui.rawui.CursorPosition = $curPos
}
```

Another good place for additional information is the console window title bar. Here is a prompt that displays the current location in the title bar to save room inside the console and still display the current location:

```
function prompt { $host.ui.rawui.WindowTitle = (Get-Location); "PS> " }
```

And this prompt function changes colors based on your notebook battery status (provided you have a battery):

```
function prompt
{
   $charge = get-wmiobject Win32_Battery |
    Measure-Object -property EstimatedChargeRemaining -average |
    Select-Object -expandProperty Average

   if ($charge -lt 25)
   {
      $color   "Red"
   } elseif ($charge     50)
   {
      $color   "Yellow"
   } else
   {
      $color   "White"
   }
   $prompttext = "PS {0} ({1}%)>"    (get-location), $charge
   Write-Host $prompttext -nonewline -foregroundcolor $color
   " "
}
```

Summary

You can use functions to create your very own new cmdlets. In its most basic form, functions are called script blocks, which execute code whenever you enter the assigned name. That's what distinguishes functions from aliases. An alias serves solely as a replacement for another command name. As such, a function can execute whatever code you want.

PBy adding parameters, you can provide the user with the option to submit additional information to your function code. Parameters can do pretty much anything that cmdlet parameters can do. They can be mandatory, optional, have a default value, or a special data type. You can even add Switch parameters to your function.

If you want your function to work as part of a PowerShell pipeline, you will need to declare the parameter that should accept pipeline input from upstream cmdlets. You will also need to move the function code into a process block so it gets executed for each incoming result.

You can play with many more parameter attributes and declarations. Try this to get a complete overview:

```
Help advanced_parameter
```

Chapter 10. Scripts

PowerShell can be used interactively and in batch mode. All the code that you entered and tested interactively can also be stored in a script file. When you run the script file, the code inside is executed from top to bottom, pretty much like if you had entered the code manually into PowerShell.

So script files are a great way of automating complex tasks that consist of more than just one line of code. Scripts can also serve as a repository for functions you create, so whenever you run a script, it defines all the functions you may need for your daily work.

Topics Covered:

Creating a Script

A PowerShell script is a plain text file with the extension ".ps1". You can create it with any text editor or use specialized PowerShell editors like the built-in "Integrated Script Environment" called "ise", or commercial products like "PowerShell Plus".

You can place any PowerShell code inside your script. When you save the script with a generic text editor, make sure you add the file extension ".ps1".

If your script is rather short, you could even create it directly from within the console by redirecting the script code to a file:

```
' "Hello world" ' > $env:temp\myscript.ps1
```

To save multiple lines to a script file using redirection, use "here-strings":

```
@'
$cutoff = (Get-Date) - (New-Timespan -hours 24)
$filename = "$env:temp\report.txt"
Get-EventLog -LogName System -EntryType Error,Warning -After $cutoff |
Format-Table -AutoSize |
Out-File $filename -width 10000

Invoke-Item $filename
'@ > $env:temp\myscript.ps1
```

Launching a Script

To actually run your script, you need to either call the script from within an existing PowerShell window, or prepend the path with "powershell.exe". So, to run the script from within PowerShell, use this:

```
& "$env:temp\myscript.ps1"
```

By prepending the call with "&", you tell PowerShell to run the script in isolation mode. The script runs in its own scope, and all variables and functions defined by the script will be automatically discarded again once the script is done. So this is the perfect way to launch a "job" script that is supposed to just "do something" without polluting your PowerShell environment with left-overs.

By prepending the call with ".", which is called "dot-sourcing", you tell PowerShell to run the script in global mode. The script now shares the scope with the callers' scope, and functions and variables defined by the script will still be available once the script is done. Use dot-sourcing if you want to debug a script (and for example examine variables), or if the script is a function library and you want to use the functions defined by the script later.

```
Powershell.exe -noprofile -executionpolicy Bypass -file %TEMP%\myscript.ps1
```

You can use this line within PowerShell as well. Since it always starts a fresh new PowerShell environment, it is a safe way of running a script in a default environment, eliminating interferences with settings and predefined or changed variables and functions.

Execution Policy - Allowing Scripts to Run

If you launched your script from outside PowerShell, using an explicit call to powershell.exe, your scripts always ran (unless you mistyped something). That's because here, you submitted the parameter -executionpolicy and turned the restriction off for the particular call.

To enable PowerShell scripts, you need to change the ExecutionPolicy. There are actually five different execution policies which you can list with this command:

```
PS > Get-ExecutionPolicy -List

                           Scope                           ExecutionPolicy
                           -----                           ---------------
                   MachinePolicy                                 Undefined
                      UserPolicy                                 Undefined
                         process                                 Undefined
                     CurrentUser                                    Bypass
                    LocalMachine                              Unrestricted
```

The first two represent group policy settings. They are set to "Undefined" unless you defined ExecutionPolicy with centrally managed group policies in which case they cannot be changed manually.

Scope "Process" refers to the current PowerShell session only, so once you close PowerShell, this setting gets lost. CurrentUser represents your own user account and applies only to you. LocalMachine applies to all users on your machine, so to change this setting you need local administrator privileges.

The effective execution policy is the first one from top to bottom that is not set to "Undefined". You can view the effective execution policy like this:

```
PS > Get-ExecutionPolicy
Bypass
```

If all execution policies are "Undefined", the effective execution policy is set to "Restricted".

Operator	Description
Restricted	Script execution is absolutely prohibited.
Default	Standard system setting normally corresponding to "Restricted".
AllSigned	Only scripts having valid digital signatures may be executed. Signatures ensure that the script comes from a trusted source and has not been altered. You'll read more about signatures later on.
RemoteSigned	Scripts downloaded from the Internet or from some other "public" location must be signed. Locally stored scripts may be executed even if they aren't signed. Whether a script is "remote" or "local" is determined by a feature called Zone Identifier, depending on whether your mail client or Internet browser correctly marks the zone. Moreover, it will work only if downloaded scripts are stored on drives formatted with the NTFS file system.
Unrestricted	PowerShell will execute any script.

Table 10.1: Execution policy setting options

Many sources recommend changing the execution policy to “RemoteSigned” to allow scripts. This setting will protect you from potentially harmful scripts downloaded from the internet while at the same time, local scripts run fine.

The mechanism behind the execution policy is just an additional safety net for you. If you feel confident that you won’t launch malicious PowerShell code because you carefully check script content before you run scripts, then it is ok to turn off this safety net altogether by setting the execution policy to “Bypass”. This setting may be required in some corporate scenarios where scripts are run off file servers that may not be part of your own domain.

If you must ensure maximum security, you can also set execution policy to “AllSigned”. Now, every single script needs to carry a valid digital signature, and if a script was manipulated, PowerShell immediately refuses to run it. Be aware that this setting does require you to be familiar with digital signatures and imposes considerable overhead because it requires you to re-sign any script once you made changes.

Invoking Scripts like Commands

To actually invoke scripts just as easily as normal commands—without having to specify relative or absolute paths and the “.ps1” file extension—pick or create a folder to store your scripts in. Next, add this folder to your “Path” environment variable. Done.

```
md $env:appdata\PSScripts
copy-item $env:temp\myscript.ps1 $env:appdata\PSScripts\myscript.ps1
$env:path += “;$env:appdata\PSScripts ”
myscript
```

Note

The changes you made to the “Path” environment variable are temporary and only valid in your current PowerShell session. To permanently add a folder to that variable, make sure you append the “Path” environment variable within your special profile script. Since this script runs automatically each time PowerShell starts, each PowerShell session automatically adds your folder to the search path. You learn more about profile scripts in a moment.

Parameters: Passing Arguments to Scripts

Scripts can receive additional arguments from the caller in much the same way as functions do (see **Chapter 9**). Simply add the param() block defining the parameters to the top of your script. You learned about param() blocks in the previous chapter.

For example, to add parameters to your event log monitoring script, try this:

```
@'
Param(
  $hours = 24,
  [Switch]
  $show
)

$cutoff = (Get-Date) - (New-Timespan -hours $hours)
$filename = "$env:temp\report.txt"
Get-EventLog -LogName System -EntryType Error,Warning -After $cutoff |
Format-Table -AutoSize |
Out-File $filename -width 10000

If ($Show) {
  Invoke-Item $filename
} else {
  Write-Warning "The report has been generated here: $filename"
}
'@ > $env:temp\myscript.ps1
```

Now you can run your script and control its behavior by using its parameters. If you copied the script to the folder that you added to your "Path" environment variable, you can even call your script without a path name, almost like a new command:

```
PS > copy-item $env:temp\myscript.ps1 $env:appdata\PSScripts\myscript.ps1
PS > myscript -hours 300
WARNING: The report has been generated here:
C:\Users\w7-pc9\AppData\Local\Temp\report.txt
PS > myscript -hours 300 -show
```

To learn more about parameters, how to make them mandatory or how to add help to your script, refer to the previous chapter. Functions and scripts share the same mechanism.

Scopes: Variable Visibility

Any variable or function you define in a script by default is scoped "local". The variable or function is visible from subscopes (like functions or nested functions or scripts called from your script). It is not visible from superscopes (like the one calling the script) unless the script was called dot-sourced.

So by default, any function or variable you define can be accessed from any other function defined at the same scope or in a subscope:

```
function A { "Here is A" }
function B { "Here is B" }

function C { A; B }

C
```

The caller of this script cannot access any function or variable, so the script will not pollute the callers context with left-over functions or variables - unless you call the script dot-sourced like described earlier in this chapter.

By prefixing variables or function names with one of the following prefixes, you can change the default behavior.

Script: use this for “shared” variables.
Global: use this to define variables or functions in the callers’ context so they stay visible even after the script finished
Private: use this to define variables or functions that only exist in the current scope and are invisible to both super- and subscopes.

Profile Scripts: Automatic Scripts

Most changes and adjustments you make to PowerShell are only temporary, and once you close and re-open PowerShell, they are lost. To make changes and adjustments persistent, use profile scripts. These scripts get executed automatically whenever PowerShell starts (unless you specify the -noprofile paramater).

The most widely used profile script is your personal profile script for the current PowerShell host. You find its path in $profile:

```
PS > $profile
C:\Users\w7-pc9\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
```

Since this profile script is specific to your current PowerShell host, the path may look different depending on your host. When you run this command from inside the ISE editor, it looks like this:

```
PS > $profile
C:\Users\w7-pc9\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.
```

If this file exists, PowerShell runs it automatically. To test whether the script exists, use Test-Path. Here is a little piece of code that creates the profile file if it not yet exists and opens it in notepad so you can add code to it:

```
PS > if (!(Test-Path $profile)) { New-Item $profile -Type File -Force | Out-Null
 }; notepad $profile
```

There are more profile scripts. $profile.CurrentUserAllHosts returns the path to the script file that automatically runs with all PowerShell hosts, so this is the file to place code in that should execute regardless of the host you use. It executes for both the PowerShell console and the ISE editor.

$profile.AllUsersCurrentHost is specific to your current host but runs for all users. To create or change this file, you need local administrator privileges. $profile.AllUsersAllHosts runs for all users on all PowerShell hosts. Again, you need local administrator privileges to create or change this file.

Signing Scripts with Digital Signatures

To guarantee that a script comes from a safe source and wasn’t manipulated, scripts can be signed with a digital signature. This signature requires a so called “Codesigning Certificate” which is a digital certificate with a private key and the explicit purpose of validating code. You can get such a certificate from your corporate IT (if they run a PKI infrastructure), or you can buy it from certificate authorities like Verisign or Thawte. You can even create your own “self-signed” certificates which are the least trustworthy alternative.

Finding Certificates

To find all codesigning certificates installed in your personal certificate store, use the virtual cert: drive:

```
Dir cert:\Currentuser\My -codeSigningCert
    directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint                                 Subject
----------                                 -------
E24D967BE9519595D7D1AC527B6449455F949C77   CN=PowerShellTestCert
```

The *-codeSigningCert* parameter ensures that only those certificates are located that are approved for the intended “code signing” purpose and for which you have a private and secret key.

If you have a choice of several certificates, pick the certificate you want to use for signing by using Where-Object:

```
$certificate = Dir cert:\CurrentUser\My |
Where-Object { $_.Subject -eq “CN=PowerShellTestCert” }
```

You can also use low-level -NET methods to open a full-featured selection dialog to pick a certificate:

```
$Store = New-Object system.security.cryptography.X509Certificates.x509Store(“My”, “CurrentUser”)
$store.Open(“ReadOnly”)
[System.Reflection.Assembly]::LoadWithPartialName(“System.Security”)
$certificate = [System.Security.Cryptography.x509Certificates.X509Certificate2UI]::SelectFromCollection
$store.Close()
$certificate
Thumbprint                                 Subject
----------                                 -------
372883FA3B386F72BCE5F475180CE938CE1B8674   CN=MyCertificate
```

Creating/Loading a New Certificate

If there is no certificate in your certificate store, you cannot sign scripts. You can then either request/purchase a codesigning certificate and install it into your personal certificate store by double-clicking it, or you can temporarily load a certificate file into memory using *Get-PfxCertificate.*

Creating Self-Signed Certificates

The key to making self-signed certificates is the Microsoft tool makecert.exe. Unfortunately, this tool can’t be downloaded separately and it may not be spread widely. You have to download it as part of a free “Software Development Kit” (SDK). Makecert.exe is in the .NET framework SDK which you can find at *http://msdn2.microsoft.com/en-us/netframework/aa731542.aspx.*

After the SDK is installed, you'll find makecert.exe on your computer and be able to issue a new code-signing certificate with a name you specify by typing the following lines:

```
$name = "PowerShellTestCert"
pushd
Cd "$env:programfiles\Microsoft Visual Studio 8\SDK\v2.0\Bin"
.\makecert.exe -pe -r -n "CN=$name" -eku 1.3.6.1.5.5.7.3.3 -ss "my"
popd
```

It will be automatically saved to the \CurrentUser\My certificate store. From this location, you can now call and use any other certificate:

```
$name = "PowerShellTestCert"
$certificate = Dir cert:\CurrentUser\My | Where-Object { $_.Subject -eq "CN=$name"}
```

Making a Certificate "Trustworthy"

Certificates you purchased from trusted certificate authorities or your own enterprise IT are considered trustworthy by default. That's because their root is listed in the "trusted root certification authorities container. You can examine these settings like this:

```
Certmgr.msc
```

Self-signed certificates are not trustworthy by default because anyone can create them. To make them trustworthy, you need to copy them into the list of trusted root certification authorities and Trusted Publishers.

Signing PowerShell Scripts

PowerShell script signatures require only two things: a valid code-signing certificate and the script that you want to sign. The cmdlet *Set-AuthenticodeSignature* takes care of the rest.

The following code grabs the first available codesigning certificate and then signs a script:

```
$certificate = @(Dir cert:CurrentUser\My -codeSigningCert -recurse)[0]
Set-AuthenticodeSignature c:\scripts\test.ps1 $certificate
```

Likewise, to sign all PowerShell scripts on a drive, use this approach:

```
Dir C:\ -filter *.ps1 -recurse -erroraction SilentlyContinue |
Set-AuthenticodeSignature -cert $certificate
```

When you look at the signed scripts, you'll see a new comment block at the end of a script.

Attention:

You cannot sign script files that are smaller than 4 Bytes, or that are saved with Big Endian Unicode. Unfortunately, the builtin script editor ISE uses just that encoding scheme to save scripts, so you may not be able to sign scripts created with ISE unless you save the scripts with a different encoding.

Checking Scripts

To check all of your scripts manually and find out whether someone has tampered with them, use Get-AuthenticodeSignature:

```
Dir c:\ -filter *.ps1 -recurse -erroraction SilentlyContinue | Get-AuthenticodeSignature
```

If you want to find only scripts that are potentially malicious, whose contents have been tampered with since they were signed *(HashMismatch)*, or whose signature comes from an untrusted certificate *(UnknownError)*, use *Where-Object* to filter your results:

```
dir c:\ -filter *.ps1 -recurse -erroraction silentlycontinue | Get-AuthenticodeSignature |
Where-Object { 'HashMismatch', 'NotSigned', 'UnknownError' -contains $_.Status }
```

Operator	Message	Description
NotSigned	The file “xyz” is not digitally signed. The script will not execute on the system. Please see “get-help about_signing” for more details.	Since the file has no digital signature, you must use Set-AuthenticodeSignature to sign the file.
UnknownError	Only scripts having valid digital signatures may be executed. Signatures ensure that the script comes from a trusted source and has not been altered. You’ll read more about signatures later on.	The used certificate is unknown. Add the certificate publisher to the trusted root certificates authorities store.
HashMismatch	File XXX check this cannot be loaded. The contents of file “…” may have been tampered because the hash of the file does not match the hash stored in the digital signature. The script will not execute on the system. Please see “get-help about_signing” for more details.	The file contents were changed. If you changed the contents yourself, resign the file.
Valid	Signature was validated.	The file contents match the signature and the signature is valid.

Table 10.3: **Status reports of signature validation and their causes**

Summary

PowerShell scripts are plain text files with a “.ps1” file extension. They work like batch files and may include any PowerShell statements.

To run a script, you need to make sure the execution policy setting is allowing the script to execute. By default, the execution policy disables all PowerShell scripts.

You can run a script from within PowerShell: specify the absolute or relative path name to the script unless the script file is stored in a folder that is part of the “Path” environment variable in which case it is sufficient to specify the script file name.

By running a script “dot-sourced” (prepending the path by a dot and a space), the script runs in the callers’ context. All variables and functions defined in the script will remain intact even once the script finished. This can be useful for debugging scripts, and it is essential for running “library” scripts that define functions you want to use elsewhere.

To run scripts from outside PowerShell, call powershell.exe and specify the script path. There are additional parameters like -noprofile which ensures that the script runs in a default powershell environment that was not changed by profile scripts.

Digital signatures ensure that a script comes from a trusted source and has not been tampered with. You can sign scripts and also verify a script signature with Set-AuthenticodeSignature and Get-AuthenticodeSignature.

Chapter 11. Error Handling

When you design a PowerShell script, there may be situations where you cannot eliminate all possible runtime errors. If your script maps network drives, there could be a situation where no more drive letters are available, and when your script performs a remote WMI query, the remote machine may not be available.

In this chapter, you learn how to discover and handle runtime errors gracefully.

Topics Covered:

Suppressing Errors

Every cmdlet has built-in error handling which is controlled by the -ErrorAction parameter. The default ErrorAction is “Continue”: the cmdlet outputs errors but continues to run.

This default is controlled by the variable $ErrorActionPreference. When you assign a different setting to this variable, it becomes the new default ErrorAction. The default ErrorAction applies to all cmdlets that do not specify an individual ErrorAction by using the parameter -ErrorAction.

To suppress error messages, set the ErrorAction to SilentlyContinue. For example, when you search the windows folder recursively for some files or folder, your code may eventually touch system folders where you have no sufficient access privileges. By default, PowerShell would then throw an exception but would continue to search through the subfolders. If you just want the files you can get your hands on and suppress ugly error messages, try this:

```
PS> Get-Childitem $env:windir -ErrorAction SilentlyContinue -recurse -filter *.log
```

Likewise, if you do not have full local administrator privileges, you cannot access processes you did not start yourself. Listing process files would produce a lot of error messages. Again, you can suppress these errors to get at least those files that you are able to access:

```
Get-Process -FileVersion -ErrorAction SilentlyContinue
```

Suppress errors with care because errors have a purpose, and suppressing errors will not solve the underlying problem. In many situations, it is invaluable to receive errors, get alarmed and act accordingly. So only suppress errors you know are benign.

NOTE: Sometimes, errors will not get suppressed despite using SilentlyContinue. If a cmdlet encounters a serious error (which is called “Terminating Error”), the error will still appear, and the cmdlet will stop and not continue regardless of your ErrorAction setting.

Whether or not an error is considered “serious” or “terminating” is solely at the cmdlet authors discretion. For example, Get-WMIObject will throw a (non-maskable) terminating error when you use -ComputerName to access a remote computer and receive an “Access Denied” error. If Get-WMIObject encounters an “RPC system not available” error because the machine you wanted to access is not online, that is considered not a terminating error, so this type of error would be successfully suppressed.

Handling Errors

To handle an error, your code needs to become aware that there was an error. It then can take steps to respond to that error. To handle errors, the most important step is setting the ErrorAction default to Stop:

```
$ErrorActionPreference = ‘Stop’
```

As an alternative, you could add the parameter -ErrorAction Stop to individual cmdlet calls but chances are you would not want to do this for every single call except if you wanted to handle only selected cmdlets errors. Changing the default ErrorAction is much easier in most situations.

The ErrorAction setting not only affects cmdlets (which have a parameter -ErrorAction) but also native commands (which do not have such a parameter and thus can only be controlled via the default setting).

Once you changed the ErrorAction to Stop, your code needs to set up an error handler to become aware of errors. There is a local error handler (try/catch) and also a global error handler (trap). You can mix both if you want.

Try/Catch

To handle errors in selected areas of your code, use the try/catch statements. They always come as pair and need to follow each other immediately. The try-block marks the area of your code where you want to handle errors. The catch-block defines the code that is executed when an error in the try-block occurs.

Take a look at this simple example:

```
'localhost', '127.0.0.1', 'storage1', 'nonexistent', 'offline' |
  ForEach-Object {
    try {
      Get-WmiObject -class Win32_BIOS -computername $_ -ErrorAction Stop |
        Select-Object __Server, Version
    }
    catch {
      Write-Warning "Error occured: $_"
    }
  }
```

It takes a list of computer names (or IP addresses) which could also come from a text file (use Get-Content to read a text file instead of listing hard-coded computer names). It then uses Foreach-Object to feed the computer names into Get-WMIObject which remotely tries to get BIOS information from these machines.

Get-WMIObject is encapsulated in a try-block and also uses the ErrorAction setting Stop, so any error this cmdlet throws will execute the catch-block. Inside the catch-block, in this example a warning is outputted. The reason for the error is available in $_ inside the catch-block.

Try and play with this example. When you remove the -ErrorAction parameter from Get-WMIObject, you will notice that errors will no longer be handled. Also note that whenever an error occurs in the try-block, PowerShell jumps to the corresponding catch-block and will not return and resume the try-block. This is why only Get-WMIObject is placed inside the try-block, not the Foreach-Object statement. So when an error does occur, the loop continues to run and continues to process the remaining computers in your list.

The error message created by the catch-block is not yet detailed enough:

```
WARNING: Error occured: The RPC server is unavailable. (Exception from HRESULT:0x800706BA)
```

You may want to report the name of the script where the error occured, and of course you'd want to output the computer name that failed. Here is a slight variant which accomplishes these tasks. Note also that in this example, the general ErrorActionPreference was set to Stop so it no longer is necessary to submit the -ErrorAction parameter to individual cmdlets:

```
'localhost', '127.0.0.1', 'storage1', 'nonexistent', 'offline' |
  ForEach-Object {
    try {
      $ErrorActionPreference = 'Stop'
      $currentcomputer = $_
      Get-WmiObject -class Win32_BIOS -computername $currentcomputer  |
        Select-Object __Server, Version
    }
    catch {
      Write-Warning ('Failed to access "{0}" : {1} in "{2}"' -f $currentcomputer, `
      $_.Exception.Message, $_.InvocationInfo.ScriptName)
    }
  }
```

This time, the warning is a lot more explicit:

```
WARNING: Failed to access “nonexistent” : The RPC server is unavailable.
(Exception from HRESULT: 0x800706BA) in “C:\Users\w7-pc9\AppData\Local\Temp\Untitled3.ps1”
```

Here, two procedures were needed: first of all, the current computer name processed by Foreach-Object needed to be stored in a new variable because the standard $_ variable is reused inside the catch-block and refers to the current error. So it can no longer be used to read the current computer name. That’s why the example stored the content of $_ in $currentcomputer before an error could occur. This way, the script code became more legible as well.

Second, inside the catch-block, $_ resembles the current error. This variable contains a complex object which contains all details about the error. Information about the cause can be found in the property Exception whereas information about the place the error occured are found in InvocationInfo.

To examine the object stored in $_, you can save it in a global variable. This way, the object remains accessible (else it would be discarded once the catch-block is processed). So when an error was handled, you can examine your test variable using Get-Member. This is how you would adjust the catch-block:

```
catch {
      $global:test = $_
      Write-Warning (‘Failed to access “{0}” : {1} in “{2}”’ -f $currentcomputer, `
      $_.Exception.Message, $_.InvocationInfo.ScriptName)
    }
  }
```

Then, once the script ran (and encountered an error), check the content of $test:

```
PS> Get-Member -InputObject $test


   TypeName: System.Management.Automation.ErrorRecord

Name                      MemberType     Definition
----                      ----------    ----------
Equals                    Method         bool Equals(System.Object obj)
GetHashCode               Method         int GetHashCode()
GetObjectData             Method         System.Void GetObjectData(System.Runtime.Seriali...
GetType                   Method         type GetType()
ToString                  Method         string ToString()
CategoryInfo              Property       System.Management.Automation.ErrorCategoryInfo C...
ErrorDetails              Property       System.Management.Automation.ErrorDetails ErrorD...
Exception                 Property       System.Exception Exception {get;}
FullyQualifiedErrorId Property           System.String FullyQualifiedErrorId {get;}
InvocationInfo            Property       System.Management.Automation.InvocationInfo Invo...
PipelineIterationInfo Property           System.Collections.ObjectModel.ReadOnlyCollectio...
TargetObject              Property       System.Object TargetObject {get;}
PSMessageDetails          ScriptProperty System.Object PSMessageDetails {get=& { Set-Stri...
```

As you see, the error information has a number of subproperties like the one used in the example. One of the more useful properties is InvocationInfo which you can examine like this:

```
PS> Get-Member -InputObject $test.InvocationInfo


   TypeName: System.Management.Automation.InvocationInfo

Name              MemberType Definition
----             ----------- ----------
Equals            Method     bool Equals(System.Object obj)
GetHashCode       Method     int GetHashCode()
GetType           Method     type GetType()
ToString          Method     string ToString()
BoundParameters   Property   System.Collections.Generic.Dictionary`2[[System.String, m...
CommandOrigin     Property   System.Management.Automation.CommandOrigin CommandOrigin ...
ExpectingInput    Property   System.Boolean ExpectingInput {get;}
HistoryId         Property   System.Int64 HistoryId {get;}
InvocationName    Property   System.String InvocationName {get;}
Line              Property   System.String Line {get;}
MyCommand         Property   System.Management.Automation.CommandInfo MyCommand {get;}
OffsetInLine      Property   System.Int32 OffsetInLine {get;}
PipelineLength    Property   System.Int32 PipelineLength {get;}
PipelinePosition  Property   System.Int32 PipelinePosition {get;}
PositionMessage   Property   System.String PositionMessage {get;}
ScriptLineNumber  Property   System.Int32 ScriptLineNumber {get;}
ScriptName        Property   System.String ScriptName {get;}
UnboundArguments  Property   System.Collections.Generic.List`1[[System.Object, mscorli...
```

It tells you all details about the place the error occured.

Using Traps

If you do not want to focus your error handler on a specific part of your code, you can also use a global error handler which is called "Trap". Actually, a trap really is almost like a catch-block without a try-block. Check out this example:

```
trap {
      Write-Warning ('Failed to access "{0}" : {1} in "{2}"' -f $currentcomputer, `
      $_.Exception.Message, $_.InvocationInfo.ScriptName)
      continue
    }

'localhost', '127.0.0.1', 'storage1', 'nonexistent', 'offline' |
  ForEach-Object {
      $currentcomputer = $_
      Get-WmiObject -class Win32_BIOS -computername $currentcomputer -ErrorAction Stop  |
        Select-Object __Server, Version
  }
```

This time, the script uses a trap at its top which looks almost like the catch-block used before. It does contain one more statement to make it act like a catch-block: Continue. Without using Continue, the trap would handle the error but then forward it on to other handlers including PowerShell. So without Continue, you would get your own error message and then also the official PowerShell error message.

When you run this script, you will notice differences, though. When the first error occurs, the trap handles the error just fine, but then the script stops. It does not execute the remaining computers in your list. Why?

Whenever an error occurs and your handler gets executed, it continues execution with the next statement following the erroneous statement - in the scope of the handler. So when you look at the example code, you'll notice that the error occurred inside the Foreach-Object loop. Whenever your code uses braces, the code inside the braces resembles a new "territory" or "scope". So the trap did process the first error correctly and then continued with the next statement in its own scope. Since there was no code following your loop, nothing else was executed.

This example illustrates that it always is a good idea to plan what you want your error handler to do. You can choose between try/catch and trap, and also you can change the position of your trap.

If you placed your trap inside the "territory" or "scope" where the error occurs, you could make sure all computers in your list are processed:

```
'localhost', '127.0.0.1', 'storage1', 'nonexistent', 'offline' |
  ForEach-Object {
      trap {
        Write-Warning ('Failed to access "{0}" : {1} in "{2}"' -f $currentcomputer, `
        $_.Exception.Message, $_.InvocationInfo.ScriptName)
        continue
      }

      $currentcomputer = $_
      Get-WmiObject -class Win32_BIOS -computername $currentcomputer -ErrorAction Stop  |
        Select-Object __Server, Version
  }
```

Handling Native Commands

Most errors in your PowerShell code can be handled in the way described above. The only command type that does not fit into this error handling scheme are native commands. Since these commands were not developed specifically for PowerShell, and since they do not necessarily use the .NET framework, they cannot directly participate in PowerShells error handling.

Console-based applications return their error messages through another mechanism: they emit error messages using the console ErrOut channel. PowerShell can monitor this channel and treat outputs that come from this channel as regular exceptions. To make this work, you need to do two things: first of all, you need to set $ErrorActionPreference to Stop, and second, you need to redirect the ErrOut channel to the StdOut channel because only this channel is processed by PowerShell. Here is an example:

When you run the following native command, you will receive an error, but the error is not red nor does it look like the usual PowerShell error messages because it comes as plain text directly from the application you ran:

```
PS> net user willibald
The user name could not be found.

More help is available by typing NET HELPMSG 2221.
```

When you redirect the error channel to the output channel, the error suddenly becomes red and is turned into a “real” PowerShell error:

```
PS> net user willibald 2>&1
net.exe : The user name could not be found.
At line:1 char:4
+ net <<<<  user willibald 2>&1
    + CategoryInfo          : NotSpecified: (The user name could not be found.:String)
   [], RemoteException
    + FullyQualifiedErrorId : NativeCommandError

More help is available by typing NET HELPMSG 2221.
```

You can still not handle the error. When you place the code in a try/catch-block, the catch-block never executes:

```
try {
  net user willibald 2>&1
  }

catch {
  Write-Warning “Oops: $_”
}
```

As you know from cmdlets, to handle errors you need to set the ErrorAction to Stop. With cmdlets, this was easy because each cmdlet has a -ErrorAction preference. Native commands do not have such a parameter. This is why you need to use $ErrorActionPreference to set the ErrorAction to Stop:

```
try {
  $ErrorActionPreference = ‘Stop’
  net user willibald 2>&1
  }

catch {
  Write-Warning “Oops: $_”
}
```

Note

If you do not like the default colors PowerShell uses for error messages, simply change them:

```
$Host.PrivateData.ErrorForegroundColor = “Red”
$Host.PrivateData.ErrorBackgroundColor = “White”
```

You can also find additional properties in the same location which enable you to change the colors of warning and debugging messages (like WarningForegroundColor and WarningBackgroundColor).

Understanding Exceptions

Exceptions work like bubbles in a fish tank. Whenever a fish gets sick, it burps, and the bubble bubbles up to the surface. If it reaches the surface, PowerShell notices the bubble and throws the exception: it outputs a red error message.

In this chapter, you learned how you can catch the bubble before it reaches the surface, so PowerShell would never notice the bubble, and you got the chance to replace the default error message with your own or take appropriate action to handle the error.

The level the fish swims in the fish tank resembles your code hierarchy. Each pair of braces resembles own “territory” or “scope”, and when a scope emits an exception (a “bubble”), all upstream scopes have a chance to catch and handle the exception or even replace it with another exception. This way you can create complex escalation scenarios.

Handling Particular Exceptions

The code set by *Trap* is by default executed for any (visible) exception. If you’d prefer to use one or several groups of different error handlers, write several *Trap* (or Catch) statements and specify for each the type of exception it should handle:

```
function Test
{
  trap [System.DivideByZeroException] { “Divided by null!”; continue }
  trap [System.Management.Automation.ParameterBindingException] {
        “Incorrect parameter!”;
        continue
         }
  1/$null
  Dir -MacGuffin
}
Test
Divided by null!
Incorrect parameter!
```

Throwing Your Own Exceptions

If you develop functions or scripts and handle errors, you are free to output error information any way you want. You could output it as plain text, use a warning or write error information to a log file. With any of these, you take away the opportunity for the caller to respond to errors - because the caller has no longer a way of detecting the error. That’s why you can also throw your own exceptions. They can be caught by the caller using a trap or a try/catch-block.

```
function TextOutput([string]$text)
{
  if ($text -eq “”)
  {
    Throw “You must enter some text.”
  }
  else
  {
    "OUTPUT: $text"
  }
}

# An error message will be thrown if no text is entered:
TextOutput
You have to enter some text.
At line:5 char:10
+      Throw  <<<< "You have to enter some text."

# No error will be output in text output:
TextOutput Hello
OUTPUT: Hello
```

The caller can now handle the error your function emitted and choose by himself how he would like to respond to it:

```
PS> try { TextOutput } catch { "Oh, an error: $_" }
Oh, an error: You must enter some text.
```

Stepping And Tracing

Commercial PowerShell development environments like PowerShellPlus from Idera make it easy for you to set breakpoints and step through code to see what it actually does. In larger scripts, this is an important diagnostic feature to debug code.

However, PowerShell has also built-in methods to step code or trace execution. To enable tracing, use this:

```
PS> Set-PSDebug -trace 1
PS> dir
DEBUG:    1+  <<<< dir
DEBUG:    1+ $_.PSParentPath.Replace <<<< ("Microsoft.PowerShell.Core\FileSystem::", "")
DEBUG:    2+                                    [String]::Format <<<< ("{0,10}  {1,8}",
 $_.LastWriteTime.ToString("d"), $_.LastWriteTime.ToString("t"))


    Directory: C:\Users\w7-pc9


Mode                LastWriteTime     Length Name
----                -------------- ------- ----
d----        30.11.2009     12:54            Application data
DEBUG:    2+                                    [String]::Format <<<< ("{0,10}  {1,8}",
 $_.LastWriteTime.ToString("d"), $_.LastWriteTime.ToString("t"))
d-r--        04.08.2010     06:36            Contacts
(...)
```

Simple tracing will show you only PowerShell statements executed in the current context. If you invoke a function or a script, only the invocation will be shown but not the code of the function or script. If you would like to see the code, turn on detailed traced by using the *-trace 2* parameter.

```
Set-PSDebug -trace 2
```

To step code, use this statement:

```
Set-PSDebug -step
```

Now, when you execute PowerShell code, it will ask you for each statement whether you want to continue, suspend or abort.

If you choose *Suspend* by pressing “H”, you will end up in a nested prompt, which you will recognize by the “<<” sign at the prompt. The code will then be interrupted so you could analyze the system in the console or check variable contents. As soon as you enter *Exit*, execution of the code will continue. Just select the “A” operation for “Yes to All” in order to turn off the stepping mode.

Tip: You can create simple breakpoints by using nested prompts: call *$host.EnterNestedPrompt()* inside a script or a function.

Note

Set-PSDebug has another important parameter called -strict. It ensures that unknown variables will throw an error. Without the Strict option, PowerShell will simply set a null value for unknown variables. On machines where you develop PowerShell code, you should enable strict mode like this:

```
Set-StrictMode -Version Latest
```

This will throw exceptions for unknown variables (possible typos), nonexistent object properties and wrong cmdlet call syntax.

Summary

To handle errors in your code, make sure you set the ErrorAction to Stop. Only then will cmdlets and native commands place errors in your control.

To detect and respond to errors, use either a local try/catch-block (to catch errors in specific regions of your code) or trap (to catch all errors in the current scope). With trap, make sure to also call Continue at the end of your error handler to tell PowerShell that you handled the error. Else, it would still bubble up to PowerShell and cause the default error messages.

To catch errors from console-based native commands, redirect their ErrOut channel to StdOut. PowerShell then automatically converts the custom error emitted by the command into a PowerShell exception.

Chapter 12. Managing Scope

Anything you define in PowerShell - variables, functions, or settings - have a certain life span. Eventually, they expire and are automatically removed from memory. This chapter talks about "scope" and how you manage the life span of objects or scripts.

Understanding and correctly managing scope can be very important. You want to make sure that a production script is not negatively influenced by "left-overs" from a previous script. Or you want certain PowerShell settings to apply only within a script. Maybe you are also wondering just why functions defined in a script you run won't show up in your PowerShell console. These questions all touch "scope".

At the end of this chapter, we will also be looking at how PowerShell finds commands and how to manage and control commands if there are ambiguous command names.

Topics Covered:

What's a Scope, Anyway

"Scope" represents the area a given object is visible in. You could also call it "territory". When you define something in one territory, another territory may not see the object. There are important default territories or scopes in PowerShell:

· PowerShell Session: Your PowerShell session - the PowerShell console or a development environment like ISE - always opens the first scope which is called "global". Anything you define in that scope persists until you close PowerShell.

· Script: When you run a PowerShell script, this script by default runs in its own scope. So any variables or functions a script declares will automatically be cleared again when the script ends. This ensures that a script will not leave behind left-overs that may influence the global scope or other scripts that you run later. Note that the default behavior can be changed both by the user and the programmer, enabling the script to store variables or functions in the callers' scope. You'll learn about that in a minute.

· Function: Every function runs yet in another scope, so variables and functions declared in a function are by default not visible to the outside. This guarantees that functions won't interfere with each other and write to the same variables - unless that is what you want. To create "shared" variables that are accessible to all functions, you would manually change scope. Again, that'll be discussed in a minute.

· Script Block: Since functions really are named script blocks, what has been said about functions also applies to script blocks. They run in their own scope or territory too.

Accessing Variables in Other Scopes

When you define a variable in your PowerShell console, you learned that it is stored in the global scope which is parent to all other scopes. Will this variable be available in child scopes, too? Let's say you are calling a script or a function. Will the variable be accessible from within the script or function?

Yes, it will. By default, anything you define in a scope is visible to all child scopes. Although it looks a bit like "inheritance", it really works different, though.

Whenever PowerShell tries to access a variable or function, it first looks in the current scope. If it is not found there, PowerShell traverses the parent scopes and continues its search until it finds the object or ultimately reaches the global scope. So, what you get will always be the variable or function that was declared in closest possible proximity to your current scope or territory.

By default, unless a variable is declared in the current scope, there is no guarantee that you access a specific variable in a specific scope. Let's assume you created a variable *$a* in the PowerShell console. When you now call a script, and the script accesses the variable $a, two things can happen: if your script has defined $a itself, you get the scripts' version of $a. If the script has not defined *$a*, you get the variable from the global scope that you defined in the console.

So here is the first golden rule that derives from this: in your scripts and functions, always declare variables and give them an initial value. If you don't, you may get unexpected results. Here is a sample:

```
function Test {
  if ($true -eq $hasrun) {
    'This function was called before'
  } else {
    $hasrun = $true
    'This function runs for the first time'
  }
}
```

When you call the function *Test* for the first time, it will state that it was called for the first time. When you call it a second time, it should notice that it was called before. In reality, the function does not, though. Each time you call it, it reports that it has been running for the first time. Moreover, in the PowerShell console enter this line:

```
$hasrun = 'some value'
```

When you now run the function *Test* again, it suddenly reports that it ran before. So the function is not at all doing what it was supposed to do. All of the unexpected behaviors can be explained with scopes.

Since each function creates its own scope, all variables defined within only exist while the function executes. Once the function is done, the scope is discarded. That's why the variable $hasrun cannot be used to remember a previous function call. Each time the function runs, a new *$hasrun* variable is created.

So why then does the function report that it has been called before once you define a variable $hasrun with arbitrary content in the console?

When the function runs, the *if* statement checks to see whether *$hasrun* is equal to *$true*. Since at that point there is no $hasrun variable in this scope, PowerShell starts to search for the variable in the parent scopes. Here, it finds the variable. And since the if statement compares a boolean value with the variable content, automatic type casting takes place: the content of the variable is automatically converted to a boolean value. Anything except *$null* will result in *$true*. Check it out, and assign *$null* to the variable, then call the function again:

```
PS> $hasrun = $null
PS> test
This function runs for the first time
```

To solve this problem and make the function work, you have to use global variables. A global variable basically is what you created manually in the PowerShell console, and you can create and access global variables programmatically, too. Here is the revised function:

```
function Test {
  if ($global:hasrun -eq $true) {
    'This function was called before'
  } else {
    $global:hasrun = $true
    'This function runs for the first time'
  }
}
```

Now the function works as expected:

```
PS> test
This function runs for the first time
PS> test
This function was called before
```

There are two changes in the code that made this happen:

- Since all variables defined inside a function have a limited life span and are discarded once the function ends, store information that continues to be present after that needs in the global scope. You do that by adding "global:" to your variable name.

- To avoid implicit type casting, reverse the order of the comparison. PowerShell always looks at the type to the left, so if that is a boolean value, the variable content will also be turned into a boolean value. As you have seen, this may result in unexpected cross-effects. By using your variable first and comparing it to *$true*, the variable type will not be changed.

Note that in place of global:, you can also use script:. That's another scope that may be useful. If you run the example in the console, they both represent the same scope, but when you define your function in a script and then run the script, script: refers to the script scope, so it creates "shared variables" that are accessible from anywhere inside the script. You will see an example of this shortly.

Keeping Information Private

Often, you want to make sure variables or functions do not spill over and pollute the global environment, so you want to make sure they are kept private. PowerShell by default does that for you, because variables and functions can only be seen by child scopes. They do not change the parent scopes.

The same is true for most PowerShell settings because they too are defined by variables. Let's take a look at the *ErrorActionPreference* setting. It determines what a cmdlet should do when it encounters a problem. By default, it is set to *'Continue'*, so PowerShell displays an error message but continues to run.

In a script, when you set *$ErrorActionPreference* to *'Stop',* you can trap errors and handle them yourself. Here is a simple example. Type in the following code and save it as a script, and then run the script:

```
$ErrorActionPreference = 'Stop'

trap {
  "Something bad occured: $_"
  continue
}

"Starting"
dir nonexisting:
Get-Process willsmith
"Done"
```

When you run this script, both errors are caught, and your script controls the error messages itself. Once the script is done, check the content of *$ErrorActionPreference:*

```
PS> $ErrorActionPreference
continue
```

It is still set to 'Continue'. By default, the change made to *$ErrorActionPreference* was limited to your script and did not change the setting in the parent scope. That's good because it prevents unwanted side-effects and left-overs from previously running scripts.

Note: If the script did change the global setting, you may have called your script "dot-sourced". We'll discuss this shortly. To follow the example, you need to call your script the default way: in the PowerShell console, enter the complete path to your script file. If you have to place the path in quotes because of spaces, prepend it with "&".

Using Private Scopes

In the previous script, the change to *$ErrorActionPreference* is automatically propagated to all child scopes. That's the default behavior. While this does not seem to be a bad thing - and in most cases is what you need - it may become a problem in complex script solutions. Just assume your script calls another script.

Now, the second script becomes a child scope, and your initial script is the parent scope. Since your initial script has changed *$ErrorActionPreference*, this change is propagated to the second script, and error handling changes there as well.

Here is a little test scenario. Type in and save this code as *script1.ps1*:

```
$ErrorActionPreference = 'Stop'

trap {
  "Something bad occured: $_"
  continue
}

$folder = Split-Path $MyInvocation.MyCommand.Definition

'Starting Script'
dir nonexisting:
'Starting Subscript'
& "$folder\script2.ps1"
'Done'
```

Now create a second script and call it *script2.ps1.* Save it in the same folder:

```
'Starting Script'
dir nonexisting:
Get-Process noprocess
"script2 ending"
```

When you run *script2.ps1*, you get two error messages from PowerShell. As you can see, the entire script2.ps1 is executed. You can see both the start message and the end message:

```
PS> & 'C:\scripts\script2.ps1'
script2 starting
Get-ChildItem : Cannot find drive. A drive with the name 'nonexisting' does not exist.
At C:\scripts\script2.ps1:2 char:4
+ dir <<<<  nonexisting:
    + CategoryInfo          : ObjectNotFound: (nonexisting:String) [Get-ChildItem], DriveNotFoundException
    + FullyQualifiedErrorId : DriveNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand
```

```
Get-Process : Cannot find a process with the name "noprocess". Verify the process name and call the cmdlet again.
At C:\scripts\script2.ps1:3 char:12
+ Get-Process <<<<  noprocess
    + CategoryInfo          : ObjectNotFound: (noprocess:String) [Get-process], ProcessCommandException
    + FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.GetProcessCommand

script2 ending
```

That is expected behavior. By default, the *ErrorActionPreference* is set to *"Continue"*, so PowerShell outputs error messages and continues with the next statement.

Now call *script1.ps1* which basically calls *script2.ps1* internally. The output suddenly is completely different:

```
PS> & 'C:\scripts\script1.ps1'
Starting Script
Something bad occured: Cannot find drive. A drive with the name 'nonexisting' does not exist.
Starting Subscript
script2 starting
Something bad occured: Cannot find drive. A drive with the name 'nonexisting' does not exist.
Done
```

No PowerShell error messages anymore. *script1.ps1* has propagated the *ErrorActionPreference* setting to the child script, so the child script now also uses the setting *"Continue"*. Any error in *script2.ps1* now bubbles up to the next available error handler which happens to be the trap in *script1.ps1.* That explains why the first error in *script2.ps1* was output by the error handler in script1.ps1.

When you look closely at the result, you will notice though that script2.ps1 was aborted. It did not continue to run. Instead, when the first error occurred, all remaining calls where skipped.

That again is default behavior: the error handler in *script1.ps1* uses the statement *"Continue"*, so after an error was reported, the error handler continues. It just does not continue in *script2.ps1*. That's because an error handler always continues with the next statement that resides in the same scope the error handler is defined. *script2.ps1* is a child scope, though.

Here are two rules that can correct the issues:

- If you want to call child scripts without propagating information or settings, make sure you mark them as private:. Note though that this will also prevent the changes from being visible in other child scopes such as functions you may have defined.

- If you do propagate *$ErrorActionPreference='Stop'* to child scripts, make sure you also implement an error handler in that script or else the script will be aborted at the first error.

    - Library Script: your script is not actually performing a task but it is rather working like a library. It defines functions for later use.

    - Debugging: you want to explore variable content after a script has run.

Here is the revised *script1.ps1* that uses private:

```
$private:ErrorActionPreference = 'Stop'

trap {
  "Something bad occured: $_"
  continue
}
```

```
$folder = Split-Path $MyInvocation.MyCommand.Definition

'Starting Script'
dir nonexisting:
'Starting Subscript'
& "$folder\script2.ps1"
'Done'
```

And this is the result:

```
PS> & 'C:\scripts\script1.ps1'
Starting Script
Something bad occured: Cannot find drive. A drive with the name 'nonexisting' does not exist.
Starting Subscript
script2 starting
Get-ChildItem : Cannot find drive. A drive with the name 'nonexisting' does not exist.
At C:\scripts\script2.ps1:2 char:4
+ dir <<<<  nonexisting:
    + CategoryInfo          : ObjectNotFound: (nonexisting:String) [Get-ChildItem], DriveNotFoundException
    + FullyQualifiedErrorId  : DriveNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

Get-Process : Cannot find a process with the name "noprocess". Verify the process name and call the cmdlet again.
At C:\scripts\script2.ps1:3 char:12
+ Get-Process <<<<  noprocess
    + CategoryInfo          : ObjectNotFound: (noprocess:String) [Get-process], ProcessCommandException
    + FullyQualifiedErrorId  : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.GetProcessCommand

script2 ending
Done
```

Now, errors in *script1.ps1* are handled by the built-in error handler, and errors in *script2.ps1* are handled by PowerShell.

And this is the revised *script2.ps1* that uses its own error handler.

```
trap {
  "Something bad occured: $_"
  continue
}

"script2 starting"
dir nonexisting:
Get-Process noprocess
"script2 ending
```

Make sure you change *script1.ps1* back to the original version by removing "private:" again before you run it:

```
PS> & 'C:\scripts\script1.ps1'
Starting Script
Something bad occured: Cannot find drive. A drive with the name 'nonexisting' does not exist.
Starting Subscript
script2 starting
Something bad occured: Cannot find drive. A drive with the name 'nonexisting' does not exist.
Something bad occured: Cannot find a process with the name "noprocess". Verify the process name and
call the cmdlet again.
script2 ending
Done
```

This time, all code in *script2.ps1* was executed and each error was handled by the new error handler in *script2.ps1.*

Calling Scripts "Dot-Sourced"

In the previous chapter you learned that a PowerShell developer can select the scope PowerShell should use to access a variable or function. The user also has control over how scoping works.

In Figure 12.1 you see that by default, the global scope (representing the PowerShell console or development environment) and the script scope (representing a script you called from global scope) are two different scopes. This guarantees that a script cannot change the caller's scope (unless the script developer used the 'global:' prefix as described earlier).

If the caller calls the script "dot-sourced", though, the script scope is omitted, and what would have been the script scope now is the global scope - or put differently, global scope and script scope become the same.

This is how you can make sure functions and variables defined in a script remain accessible even after the script is done. Here is a sample. Type in the code and save it as *script3.ps1:*

```
function test-function {
  'I am a test function!'
}

test-function
```

When you run this script the default way, the function *test-function* runs once because it is called from within the script. Once the script is done, the function is gone. You can no longer call *test-function.*

```
PS> & 'C:\script\script3.ps1'
I am a test function!
PS> test-function
The term 'test-function' is not recognized as the name of a cmdlet, function, script file, or operable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:14
+ test-function <<<<
    + CategoryInfo          : ObjectNotFound: (test-function:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

Now, run the script dot-sourced! You do that by replacing the call operator "&" by a dot:

```
PS> . 'C:\script\script3.ps1'
I am a test function!
PS> test-function
I am a test function!
```

Since now the script scope and the global scope are identical, the script did define the function *test-function* in the global scope. That's why the function is still there once the script ended.

There are two primary reasons to use dot-sourcing:

The profile script that PowerShell runs automatically during startup (*$profile*) is an example of a script that is running dot-sourced, although you cannot see the actual dot-sourcing call.

Note: To make sure functions defined in a script remain accessible, a developer could also prepend the function name with "global:". However, that may not be such a clever idea. The prefix "global:" always creates the function in the global context. Dot-sourcing is more flexible because it creates the function in the caller's context. So if a script runs another script dot-sourced, all functions defined in the second script are also available in the first, but the global context (the console) remains unaffected and unpolluted.

Managing Command Types

PowerShell supports a wide range of command types, and when you call a command, there is another type of scope. Each command type lives in its own scope, and when you ask PowerShell to execute a command, it searches the command type scopes in a specific order.

This default behavior is completely transparent if there is no ambiguity. If however you have different command types with the same name, this may lead to surprising results:

```
# Run an external command:
ping -n 1 10.10.10.10
Pinging 10.10.10.10 with 32 bytes of data:
Reply from 10.10.10.10: Bytes=32 Time<1ms TTL=128
Ping statistics for 10.10.10.10:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% Loss),
Ca. time in millisec:
    Minimum = 2ms, Maximum = 2ms, Average = 2ms

# Create a function having the same name:
function Ping { "Ping is not allowed." }

# Function has priority over external program and turns off command:
ping -n 1 10.10.10.10
Ping is not allowed.
```

Now, Ping calls the *Echo* command, which is an alias for *Write-Output* and simply outputs the parameters that you may have specified after Ping in the console.

CommandType	Description	Priority
Alias	An alias for another command added by using Set-Alias	1
Function	A PowerShell function defined by using function	2
Filter	A PowerShell filter defined by using filter (a function with a process block)	2
Cmdlet	A PowerShell cmdlet from a registered snap-in	3
Application	An external Win32 application	4
ExternalScript	An external script file with the file extension ".ps1"	5
Script	A scriptblock	-

Table 12.1: Various PowerShell command types

Get-Command can tell you whether there are ambiguities:

```
Get-Command Ping
CommandType     Name                                        Definition
-----------     ----                                        ----------
function        Ping                                       "Ping is not allowed."
Alias           ping                                        echo
Application     PING.EXE                                   C:\Windows\system32\PING.EXE
```

Invoking a Specific Command Type

To make sure you invoke the command type you are after, you can use Get-Command to retrieve the command type, and then execute it with the call operator "&". So in the example above, to explicitly call *ping.exe*, use this:

```
# Get command named "Ping" with commandtype "Application":
$command = Get-Command Ping -CommandType Application

# Call the command
& $command -n 1 10.10.10.10
Pinging 10.10.10.10 with 32 bytes of data:
Reply from 10.10.10.10: Bytes=32 Time<1ms TTL=128
Ping statistics for 10.10.10.10:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% Loss),
Ca. time in millisec:
    Minimum = 2ms, Maximum = 2ms, Average = 2ms
```

Summary

PowerShell uses scopes to manage the life span and visibility of variables and functions. By default, the content of scopes is visible to all child scopes and does not change any parent scope.

There is always at least one scope which is called “global scope”. New scopes are created when you define scripts or functions.

The developer can control the scope to use by prepending variable and function names with one of these keywords: global:, script:, private: and local:. The prefix local: is the default and can be omitted.

The user can control scope by optionally dot-sourcing scripts, functions or script blocks. With dot sourcing, for the element you are calling, no new scope is created. Instead, the caller’s context is used.

A different flavor of scope is used to manage the five different command types PowerShell supports. Here, PowerShell searches for commands in a specific order. If the command name is ambiguous, PowerShell uses the first command it finds. To find the command, it searches the command type scopes in this order: alias, function, cmdlet, application, external script, and script. Use Get-Command to locate a command yourself based on name and command type if you need more control.

Chapter 13.
Text and Regular Expressions

Often, you need to deal with plain text information. You may want to read the content from some text file and extract lines that contain a keyword, or you would like to isolate the file name from a file path. So while the object-oriented approach of PowerShell is a great thing, at the end of a day most useful information breaks down to plain text. In this chapter, you'll learn how to control text information in pretty much any way you want.

Topics Covered:

Defining Text

To define text, place it in quotes. If you want PowerShell to treat the text exactly the way you type it, use single quotes. Use double quotes with care because they can transform your text: any variable you place in your text will get resolved, and PowerShell replaces the variable with its context. Have a look:

```
$text = 'This text may also contain $env:windir `: $(2+2)'
This text may also contain $env:windir `: $(2+2)
```

Placed in single quotes, PowerShell returns the text exactly like you entered it. With double quotes, the result is completely different:

```
$text = "This text may also contain $env:windir `: $(2+2)"
This text may also contain C:\Windows: 4
```

Special Characters in Text

The most common "special" character you may want to put in text are quotes. Quotes are tricky because you need to make sure that PowerShell does not confuse the quotes inside your text with the quotes that actually surround and define the text. You do have a couple of choices.

If you used single quotes to delimit the text, you can freely use double quotes inside the text, and vice versa:

```
'The "situation" was really not that bad'
"The 'situation' was really not that bad"
```

If you must use the same type of quote both as delimiter and inside the text, you can "escape" quotes (remove their special meaning) by either using two consecutive quotes, or by placing a "backtick" character in front of the quote:

```
'The ''situation'' was really not that bad'
"The ""situation"" was really not that bad"
'The `'situation`' was really not that bad'
"The `"situation`" was really not that bad"
```

The second most wanted special character you may want to include in text is a new line so you can extend text to more than one line. Again, you have a couple of choices.

When you use double quotes to delimit text, you can insert special control characters like tabs or line breaks by adding a backtick and then a special character where "t" stands for a tab and "n" represents a line break. This technique does require that the text is defined by double quotes:

```
PS> "One line`nAnother line"
One line
Another line
PS> 'One line`nAnother line'
One line`nAnother line
```

Escape Sequence	Special Characters
`n	New line
`r	Carriage return
`t	Tabulator
`a	Alarm
`b	Backspace
`'	Single quotation mark
`"	Double quotation mark
`0	Null
``	Backtick character

Table 13.1: Special characters and “escape” sequences for text

Resolving Variables

A rather unusual special character is “$”. PowerShell uses it to define variables that can hold information. Text in double quotes also honors this special character and recognizes variables by resolving them: PowerShell automatically places the variable content into the text:

```
$name = 'Weltner'
"Hello Mr $name"
```

This only works for text enclosed in double quotes. If you use single quotes, PowerShell ignores variables and treats “$” as a normal character:

```
"Hello Mr $name"
```

At the same time, double quotes protect you from unwanted variable resolving. Take a look at this example:

```
"My wallet is low on $$$$"
```

As turns out, $$ is again a variable (it is an internal “automatic” variable maintained by PowerShell which happens to contain the last command token PowerShell processed which is why the result of the previous code line can vary and depends on what you executed right before), so as a rule of thumb, you should start using single quotes by default unless you really want to resolve variables in your text. Resolving text can be enormously handy:

```
PS> $name = "report"
PS> $extension = "txt"
PS> "$name.$extension"
report.txt
```

Just make sure you use it with care.

Now, what would you do if you needed to use “$” both to resolve variables and to display literally in the same text? Again, you can use the backtick to escape the “$” and remove its special resolving capability:

```
"The variable `$env:windir contains ""$env:windir"""
```

Tip: You can use the “$” resolving capabilities to insert live code results into text. Just place the code you want to evaluate in brackets. To make PowerShell treat these brackets as it would outside of text, place a “$” before:

```
$result = "One CD has the capacity of $(720MB / 1.44MB) diskettes."
$result
One CD has the capacity of 500 diskettes.
```

“Here-Strings”: Multi-Line Text

As you have seen, you can insert special backtick-key-combinations to insert line breaks and produce multi-line text. While that may work for one or two lines of text, it quickly becomes confusing for the reader and tiresome for the script author to construct strings like that.

A much more readable way is using here-strings. They work like quotes except they use a “@” before and after the quote to indicate that the text extends over multiple lines.

```
$text = @"
>> Here-Strings can easily stretch over several lines and may also include
>>"quotation marks". Nevertheless, here, too, variables are replaced with
>> their values: C:\Windows, and subexpressions like 4  are likewise replaced
>> with their result. The text will be concluded only if you terminate the
>> here-string with the termination symbol "@.
>> "@
>>
$text
Here-Strings can easily stretch over several lines and may also include
"quotation marks". Nevertheless, here, too, variables are replaced with
their values: C:\Windows, and subexpressions like 4  are likewise replaced
with their result. The text will be concluded only if you terminate the
here-string with the termination symbol "@.
```

Communicating with the User

Maybe you don’t want to hard-code text information in your script at all but instead provide a way for the user to enter information. To accept plain text input use *Read-Host:*

```
$text = Read-Host "Enter some text"
Enter some text: Hello world!
$text
Hello world!
```

Text accepted by *Read-Host* is treated literally, so it behaves like text enclosed in single quotes. Special characters and variables are not resolved. If you want to resolve the text a user entered, you can however send it to the internal *ExpandString()* method for post-processing. PowerShell uses this method internally when you define text in double quotes:

```
# Query and output text entry by user:
$text = Read-Host "Your entry"
Your entry: $env:windir
$text
$env:windir

# Treat entered text as if it were in double quotation marks:
$ExecutionContext.InvokeCommand.ExpandString($text)

C:\Windows
```

You can also request secret information from a user. To mask input, use the switch parameter *-asSecureString*. This time, however, *Read-Host* won't return plain text anymore but instead an encrypted *SecureString*. So, not only the input was masked with asterisks, the result is just as unreadable. To convert an encrypted *SecureString* into plain text, you can use some internal .NET methods:

```
$pwd = Read-Host -asSecureString "Password"
Password: *************
$pwd
System.Security.SecureString
[Runtime.InteropServices.Marshal]::PtrToStringAuto([Runtime.InteropServices.Marshal]::SecureStringToBSTR($pwd))
strictly confidential
```

Composing Text with "-f"

The *–f* format operator is the most important PowerShell string operator. You'll soon be using it to format numeric values for easier reading:

```
"{0:0} diskettes per CD" -f (720mb/1.44mb)
500 diskettes per CD
```

The -f format operator formats a string and requires a string, along with wildcards on its left side and on its right side, that the results are to be inserted into the string instead of the wildcards:

```
"{0} diskettes per CD" -f (720mb/1.44mb)
500 diskettes per CD
```

It is absolutely necessary that exactly the same results are on the right side that are to be used in the string are also on the left side. If you want to just calculate a result, then the calculation should be in parentheses. As is generally true in PowerShell, the parentheses ensure that the enclosed statement is evaluated first and separately and that subsequently, the result is processed instead of the parentheses. Without parentheses, *-f* would report an error:

```
"{0} diskettes per CD" -f 720mb/1.44mb
Bad numeric constant: 754974720 diskettes per CD.
At line:1 char:33
+ "{0} diskettes per CD" -f 720mb/1 <<<< .44mb
```

You may use as many wildcard characters as you wish. The number in the braces states which value will appear later in the wildcard and in which order:

```
"{0} {3} at {2}MB fit into one CD at {1}MB" -f (720mb/1.44mb), 1.44, 720, "diskettes"
500 diskettes at 720MB fit into one CD at 1.44MB
```

Setting Numeric Formats

The *–f* format operator can insert values into text as well as format the values. Every wildcard used has the following formal structure: *{index[,alignment][:format]}:*

- **Index:** This number indicates which value is to be used for this wildcard. For example, you could use several wildcards with the same index if you want to output one and the same value several times, or in various display formats. The index number is the only obligatory specification. The other two specifications are voluntary.
- **Alignment:** Positive or negative numbers can be specified that determine whether the value is right justified (positive number) or left justified (negative number). The number states the desired width. If the value is wider than the specified width, the specified width will be ignored. However, if the value is narrower than the specified width, the width will be filled with blank characters. This allows columns to be set flush.
- **Format:** The value can be formatted in very different ways. Here you can use the relevant format name to specify the format you wish. You'll find an overview of available formats below.

Important

Formatting statements are case sensitive in different ways than what is usual in PowerShell. You can see how large the differences can be when you format dates:

```
# Formatting with a small letter d:
"Date: {0:d}" -f (Get-Date)
Date: 08/28/2007

# Formatting with a large letter D:
"Date: {0:D}" -f (Get-Date)
Date: Tuesday, August 28, 2007
```

Symbol	Type	Call	Result
#	Digit placeholder	"{0:(#).##}" -f $value	(1000000)
%	Percentage	"{0:0%}" -f $value	100000000%
,	Thousands separator	"{0:0,0}" -f $value	1,000,000
,.	Integral multiple of 1,000	"{0:0,.} " -f $value	1000
.	Decimal point	"{0:0.0}" -f $value	1000000.0
0	0 placeholder	"{0:00.0000}" -f $val-ue	1000000.0000
c	Currency	"{0:c}" -f $value	1,000,000.00 €
d	Decimal	"{0:d}" -f $value	1000000
e	Scientific notation	"{0:e}" -f $value	1.000000e+006
e	Exponent wildcard	"{0:00e+0}" -f $value	10e+5

Symbol	Type	Call	Result
f	Fixed point	“{0:f}” -f $value	(1000000)
g	General	“{0:g}” -f $value	100000000%
n	Thousands separator	“{0:n}” -f $value	1,000,000
x	Hexadecimal	“0x{0:x4}” -f $value	1000

Table 13.3: Formatting numbers

Using the formats in Table 13.3, you can format numbers quickly and comfortably. No need for you to squint your eyes any longer trying to decipher whether a number is a million or 10 million:

```
10000000000
"{0:N0}" -f 10000000000
10,000,000,000
```

There’s also a very wide range of time and date formats. The relevant formats are listed in Table 13.4 and their operation is shown in the following lines:

```
$date= Get-Date
foreach ($format in "d","D","f","F","g","G","m","r","s","t","T","u","U","y",`
"dddd, MMMM dd yyyy","M/yy","dd-MM-yy") {
"DATE with $format : {0}" -f $date.ToString($format)
}
DATE with d : 10/15/2007
DATE with D : Monday, 15 October, 2007
DATE with f : Monday, 15 October, 2007 02:17 PM
DATE with F : Monday, 15 October, 2007 02:17:02 PM
DATE with g : 10/15/2007 02:17
DATE with G : 10/15/2007 02:17:02
DATE with m : October 15
DATE with r : Mon, 15 Oct 2007 02:17:02 GMT
DATE with s : 2007-10-15T02:17:02
DATE with t : 02:17 PM
DATE with T : 02:17:02 PM
DATE with u : 2007-10-15 02:17:02Z
DATE with U : Monday, 15 October, 2007 00:17:02
DATE with y : October, 2007
DATE with dddd, MMMM dd yyyy : Monday, October 15 2007
DATE with M/yy : 10/07
DATE with dd-MM-yy : 15-10-07
```

Symbol	Type	Call	Result
d	Short date format	"{0:d}" -f $value	09/07/2007
D	Long date format	"{0:D}" -f $value	Friday, September 7, 2007
t	Short time format	"{0:t}" -f $value	10:53:56 AM
T	Long time format	"{0:T}" -f $value	10:53:56 AM
f	Full date and time (short)	"{0:f}" -f $value	Friday, September 7, 2007 10:53 AM
F	Full date and time (long)	"{0:F}" -f $value	Friday, September 7, 2007 10:53:56 AM
g	Standard date (short)	"{0:g}" -f $value	09/07/2007 10:53 AM
G	Standard date (long)	"{0:d}" -f $value	09/07/2007 10:53:56 AM
M	Day of month	"{0:M}" -f $value	September 07
r	RFC1123 date format	"{0:r}" -f $value	Fri, 07 Sep 2007 10:53:56 GMT
s	Sortable date format	"{0:s}" -f $value	2007-09-07T10:53:56
u	Universally sortable date format	"{0:u}" -f $value	2007-09-07 10:53:56Z
U	Universally sortable GMT date format	"{0:U}" -f $value	Friday, September 7, 2007 08:53:56
Y	Year/month format pattern	"{0:Y}" -f $value	September 2007

Table 13.4: Formatting date values

If you want to find out which type of formatting options are supported, you need only look for .NET types that support the *toString()* method:

```
[AppDomain]::CurrentDomain.GetAssemblies() | ForEach-Object {
   $_.GetExportedTypes() | Where-Object {! $_.IsSubclassOf([System.Enum])}
  } | ForEach-Object {
     $Methods = $_.GetMethods() | Where-Object {$_.Name -eq "tostring"} |%{"$_"};
     if ($methods -eq "System.String ToString(System.String)") {
           $_.FullName
     }
   }
System.Enum
System.DateTime
System.Byte
System.Convert
System.Decimal
System.Double
System.Guid
System.Int16
System.Int32
System.Int64
System.IntPtr
System.SByte
System.Single
System.UInt16
System.UInt32
System.UInt64
Microsoft.PowerShell.Commands.MatchInfo
```

For example, among the supported data types is the “globally unique identifier” *System.Guid*. Because you’ll frequently require GUID, which is clearly understood worldwide, here’s a brief example showing how to create and format a GUID:

```
$guid = [GUID]::NewGUID()
foreach ($format in ”N”,”D”,”B”,”P”) {“GUID with $format : {0}” -f $GUID.ToString($format)}
GUID with N : 0c4d2c4c8af84d198b698e57c1aee780
GUID with D : 0c4d2c4c-8af8-4d19-8b69-8e57c1aee780
GUID with B : {0c4d2c4c-8af8-4d19-8b69-8e57c1aee780}
GUID with P : (0c4d2c4c-8af8-4d19-8b69-8e57c1aee780)
```

Symbol	Type	Call	Result
dd	Day of month	“{0:dd}” -f $value	07
ddd	Abbreviated name of day	“{0:ddd}” -f $value	Fri
dddd	Full name of day	“{0:dddd}” -f $value	Friday
gg	Era	“{0:gg}” -f $value	A. D.
hh	Hours from 01 to 12	“{0:hh}” -f $value	10
HH	Hours from 0 to 23	“{0:HH}” -f $value	10
mm	Minute	“{0:mm}” -f $value	53
MM	Month	“{0:MM}” -f $value	09
MMM	Abbreviated month name	“{0:MMM}” -f $value	Sep
MMMM	Full month name	“{0:MMMM}” -f $value	September
ss	Second	“{0:ss}” -f $value	56
tt	AM or PM	“{0:tt}” -f $value	
yy	Year in two digits	“{0:yy}” -f $value	07
yyyy	Year in four digits	“{0:YY}” -f $value	2007
zz	Time zone including leading zero	“{0:zz}” -f $value	+02

Table 13.5: Customized date value formats

Outputting Values in Tabular Form: Fixed Width

To display the output of several lines in a fixed-width font and align them one below the other, each column of the output must have a fixed width. A format operator can set outputs to a fixed width.

In the following example, Dir returns a directory listing, from which a subsequent loop outputs file names and file sizes. Because file names and sizes vary, the result is ragged right and hard to read:

```
dir | ForEach-Object { “$($_.name) = $($_.Length) Bytes” }
history.csv = 307 Bytes
info.txt = 8562 Bytes
layout.lxy = 1280 Bytes
list.txt = 164186 Bytes
p1.nrproj = 5808 Bytes
ping.bat = 116 Bytes
SilentlyContinue = 0 Bytes
```

The following result with fixed column widths is far more legible. To set widths, add a comma to the sequential number of the wildcard and after it specify the number of characters available to the wildcard. Positive numbers will set values to right alignment, negative numbers to left alignment:

```
dir | ForEach-Object { "{0,-20} = {1,10} Bytes" -f $_.name, $_.Length }
history.csv = 307 Bytes
info.txt = 8562 Bytes
layout.lxy = 1280 Bytes
list.txt = 164186 Bytes
p1.nrproj = 5808 Bytes
ping.bat = 116 Bytes
SilentlyContinue = 0 Bytes
```

More options are offered by special text commands that PowerShell furnishes from three different areas:

- **String operators:** PowerShell includes a number of string operators for general text tasks which you can use to replace text and to compare text (Table 13.2).
- **Dynamic methods:** the String data type, which saves text, includes its own set of text statements that you can use to search through, dismantle, reassemble, and modify text in diverse ways (Table 13.6).
- **Static methods:** finally, the String .NET class includes static methods bound to no particular text.

String Operators

All string operators work in basically the same way: they take data from the left and the right and then do something with them. The –replace operator for example takes a text and some replacement text and then replaces the replacement text in the original text:

```
"Hello Carl" -replace "Carl", "Eddie"
Hello Eddie
```

The format operator -f works in exactly the same way. You heard about this operator at the beginning of this chapter. It takes a static string template with placeholders and an array with values, and then fills the values into the placeholders.

Two additional important string operators are *-join and -split*. They can be used to automatically join together an array or to split a text into an array of substrings.

Let's say you want to output information that really is an array of information. When you query WMI for your operating system to identify the installed MUI languages, the result can be an array (when more than one language is installed). So, this line produces an incomplete output:

You would have to join the array to one string first using *-join*. Here is how:

```
PS> $mui = Get-WmiObject Win32_OperatingSystem | Select-Object -ExpandProperty MuiLanguages
PS> 'Installed MUI-Languages: {0}' -f ($mui -join ', ')
Installed MUI-Languages: de-DE, en-US
```

The *-split* operator does the exact opposite. It takes a text and a split pattern, and each time it discovers the split pattern, it splits the original text in chunks and returns an array. This example illustrates how you can use *-split* to parse a path:

```
PS> ('c:\test\folder\file.txt' -split '\\')[-1]
file.txt
```

Note that *-replace* expects the pattern to be a regular expression, so if your pattern is composed of reserved characters (like the backslash), you have to escape it. Note also that the Split-Path cmdlet can split paths more easily.

To auto-escape a simple text pattern, use .NET methods. The *Escape()* method takes a simple text pattern and returns the escaped version that you can use wherever a regular expression is needed:

```
PS> [RegEx]::Escape('some.\pattern')
some\.\\pattern
```

String Object Methods

You know from **Chapter 6** that PowerShell represents everything as objects and that every object contains a set of instructions known as methods. Text is stored in a *String object*, and a string object has built-in methods for manipulating the text information. Simply add a "." and then the method you need:

```
$path = "c:\test\Example.bat"
$path.Substring( $path.LastIndexOf(".")+1 )
bat
```

Another approach uses the dot as separator and *Split()* to split up the path into an array. The result is that the last element of the array (-1 index number) will include the file extension:

```
$path.Split(".")[-1]
bat
```

Function	Description	Example
CompareTo()	Compares one string to another	("Hello").CompareTo("Hello")
Contains()	Returns "True" if a specified comparison string is in a string or if the comparison string is empty	("Hello").Contains("ll")
CopyTo()	Copies part of a string to another string	$a = ("Hello World").toCharArray() ("User!").CopyTo(0, $a, 6, 5) $a
EndsWith()	Tests whether the string ends with a specified string	("Hello").EndsWith("lo")
Equals()	Tests whether one string is identical to another string	("Hello").Equals($a)
IndexOf()	Returns the index of the first occurrence of a comparison string	("Hello").IndexOf("l")
IndexOfAny()	Returns the index of the first occurrence of any character in a comparison string	("Hello").IndexOfAny("loe")
Insert()	Inserts new string at a specified index in an existing string	("Hello World").Insert(6, "brave ")
GetEnumerator()	Retrieves a new object that can enumerate all characters of a string	("Hello").GetEnumerator()
LastIndexOf()	Finds the index of the last occurrence of a specified character	("Hello").LastIndexOf("l")
LastIndexOfAny()	Finds the index of the last occurrence of any character of a specified string	("Hello").LastIndexOfAny("loe")
PadLeft()	Pads a string to a specified length and adds blank characters to the left (right-aligned string)	("Hello").PadLeft(10)
PadRight()	Pads string to a specified length and adds blank characters to the right (left-aligned string)	("Hello").PadLeft(10)
Remove()	Removes any requested number of characters starting from a specified position	("Hello").PadRight(10) + "World!"
Replace()	Replaces a character with another character	("Hello World").Remove(5,6)
Split()	Converts a string with specified splitting points into an array	("Hello World").toCharArray()
StartsWith()	Tests whether a string begins with a specified character	("Hello World").StartsWith("He")
Substring()	Extracts characters from a string	("Hello World").Substring(4, 3)
ToCharArray()	Converts a string into a character array	("Hello World").toCharArray()
ToLower()	Converts a string to lowercase	("Hello World").toLower()
ToLowerInvariant()	Converts a string to lowercase using casing rules of the invariant language	("Hello World").toLowerInvariant()
ToUpper()	Converts a string to uppercase	("Hello World").toUpper()

Function	Description	Example
ToUpperInvariant()	Converts a string to uppercase using casing rules of the invariant language	("Hello World").ToUpperInvariant()
Trim()	Removes blank characters to the right and left	(" Hello ").Trim() + "World"
TrimEnd()	Removes blank characters to the right	(" Hello ").TrimEnd() + "World"
TrimStart()	Removes blank characters to the left	(" Hello ").TrimStart() + "World"
Chars()	Provides a character at the specified position	("Hello").Chars(0)

Table 13.6: The methods of a string object

Analyzing Methods: Split() as Example

You already know in detail from **Chapter 6** how to use *Get-Member* to find out which methods an object contains and how to invoke them. Just as a quick refresher, let's look again at an example of the *Split()* method to see how it works.

```
("something" | Get-Member Split).definition
System.String[] Split(Params Char[] separator), System.String[] Split(Char[] separator,
 Int32 count), System.String[] Split(Char[] separator, StringSplitOptions options),
 System.String[] Split(Char[] separator, Int32 count, StringSplitOptions options),
 System.String[] Split(String[] separator, StringSplitOptions options),
 System.String[] Split(String[] separator, Int32 count, StringSplitOptions options)
```

Definition gets output, but it isn't very easy to read. Because *Definition* is also a string object, you can use methods from Table 13.6, including *Replace()*, to insert a line break where appropriate. That makes the result much more understandable:

```
("something" | Get-Member Split).Definition.Replace("), ", ")`n")
System.String[] Split(Params Char[] separator)
System.String[] Split(Char[] separator, Int32 count)
System.String[] Split(Char[] separator, StringSplitOptions options)
System.String[] Split(Char[] separator, Int32 count, StringSplitOptions options)
System.String[] Split(String[] separator, StringSplitOptions options)
System.String[] Split(String[] separator, Int32 count, StringSplitOptions options)
```

There are six different ways to invoke *Split()*. In simple cases, you might use *Split()* with only one argument, *Split()*, you will expect a character array and will use every single character as a possible splitting separator. That's important because it means that you may use several separators at once:

```
"a,b;c,d;e;f".Split(",;")
a
b
c
d
e
f
```

If the splitting separator itself consists of several characters, then it has got to be a string and not a single *Char* character. There are only two signatures that meet this condition:

```
System.String[] Split(String[] separator, StringSplitOptions options)
System.String[] Split(String[] separator, Int32 count, StringSplitOptions options)
```

You must make sure that you pass data types to the signature that is exactly right for it to be able to use a particular signature. If you want to use the first signature, the first argument must be of the *String[]* type and the second argument of the *StringSplitOptions* type. The simplest way for you to meet this requirement is by assigning arguments first to a strongly typed variable. Create the variable of exactly the same type that the signature requires:

```
# Create a variable of the [StringSplitOptions] type:
[StringSplitOptions]$option = "None"

# Create a variable of the String[] type:
[string[]]$separator = ",;"
# Invoke Split with the wished signature and use a two-character long separator:
("a,b;c,;d,e;f,;g").Split($separator, $option)
a,b;c
d,e;f
g
```

Split() in fact now uses a separator consisting of several characters. It splits the string only at the points where it finds precisely the characters that were specified. There does remain the question of how do you know it is necessary to assign the value *"None"* to the *StringSplitOptions* data type. The simple answer is: you don't know and it isn't necessary to know. If you assign a value to an unknown data type that can't handle the value, the data type will automatically notify you of all valid values:

```
[StringSplitOptions]$option = "werner wallbach"
Cannot convert value "werner wallbach" to type "System.StringSplitOptions" due to invalid
 enumeration values. Specify one of the following enumeration values and try again.
 The possible enumeration values are "None, RemoveEmptyEntries".
At line:1 char:28
+ [StringSplitOptions]$option  <<<< = "werner wallbach"
```

By now it should be clear to you what the purpose is of the given valid values and their names. For example, what was *RemoveEmptyEntries()* able to accomplish? If *Split()* runs into several separators following each other, empty array elements will be the consequence. *RemoveEmptyEntries()* deletes such empty entries. You could use it to remove redundant blank characters from a text:

```
[StringSplitOptions]$option = "RemoveEmptyEntries"
"This   text   has   too   much   whitespace".Split(" ", $option)
This
text
has
too
```

Now all you need is just a method that can convert the elements of an array back into text. The method is called *Join()*; it is not in a *String* object but in the *String* class.

Simple Pattern Recognition

Recognizing patterns is a frequent task that is necessary for verifying user entries, such as to determine whether a user has given a valid network ID or valid e-mail address.

A simple form of wildcards was invented for the file system many years ago and it still works today. In fact, you've probably used it before in one form or another:

```
# List all files in the current directory that have the txt file extension:
Dir *.txt

# List all files in the Windows directory that begin with "n" or "w":
dir $env:windir\[nw]*.*

# List all files whose file extensions begin with "t" and which are exactly 3 characters long:
Dir *.t??

# List all files that end in one of the letters from "e" to "z"
dir *[e-z].*
```

Wildcard	Description	Example
*	Any number of any character (including no characters at all)	Dir *.txt
?	Exactly one of any characters	Dir *.??t
[xyz]	One of specified characters	Dir [abc]*.*
[x-z]	One of the characters in the specified area	Dir *[p-z].*

Table 13.7: Using simple placeholders

The placeholders in **Table 13.7** work in the file system, but also with string comparisons like *-like* and *-notlike*. For example, if you want to verify whether a user has given a valid IP address, you could do so in the following way:

```
$ip = Read-Host "IP address"
if ($ip -like "*.*.*.*") { "valid" } else { "invalid" }
```

If you want to verify whether a valid e-mail address was entered, you could check the pattern like this:

```
$email = "tobias.weltner@powershell.de"
$email -like "*.*@*.*"
```

These simple patterns are not very exact, though:

```
# Wildcards are appropriate only for very simple pattern recognition and leave room for erroneous entries:
$ip = "300.werner.6666."
if ($ip -like "*.*.*.*") { "valid" } else { "invalid" }
valid

# The following invalid e-mail address was not identified as false:
$email = ".@."
$email -like "*.*@*.*"
True
```

Regular Expressions

Use regular expressions for more accurate pattern recognition. Regular expressions offer highly specific wildcard characters; that's why they can describe patterns in much greater detail. For the very same reason, however, regular expressions are also much more complicated.

Describing Patterns

Using the regular expression elements listed in **Table 13.11**, you can describe patterns with much greater precision. These elements are grouped into three categories:

- **Placeholder:** The placeholder represents a specific type of data, for example a character or a digit.
- **Quantifier:** Allows you to determine how often a placeholder occurs in a pattern. You could, for example, define a 3-digit number or a 6-character-word.
- **Anchor:** Allows you to determine whether a pattern is bound to a specific boundary. You could define a pattern that needs to be a separate word or that needs to begin at the beginning of the text.

The pattern represented by a regular expression may consist of four different character types:

- **Literal characters** like "abc" that exactly matches the "abc" string.
- **Masked or "escaped" characters with special meanings in regular expressions; when preceded by "\", they are understood as literal characters: "\[test\]" looks for the "[test]" string. The following characters have special meanings and for this reason must be masked if used literally: ". ^ $ * + ? { [ ] \ | ( )".**
- **Pre-defined wildcard characters** that represent a particular character category and work like placeholders. For example, "\d" represents any number from 0 to 9.
- **Custom wildcard characters:** They consist of square brackets, within which the characters are specified that the wildcard represents. If you want to use any character except for the specified characters, use "^" as the first character in the square brackets. For example, the placeholder "[^f-h]" stands for all characters except for "f", "g", and "h".

Element	Description
.	Exactly one character of any kind except for a line break (equivalent to [^\n])
[^abc]	All characters except for those specified in brackets
[^a-z]	All characters except for those in the range specified in the brackets
[abc]	One of the characters specified in brackets
[a-z]	Any character in the range indicated in brackets
\a	Bell alarm (ASCII 7)
\c	Any character allowed in an XML name
\cA-\cZ	Control+A to Control+Z, equivalent to ASCII 0 to ASCII 26
\d	A number (equivalent to [0-9])
\D	Any character except for numbers
\e	Escape (ASCII 9)
\f	Form feed (ASCII 15)
\n	New line
\r	Carriage return
\s	Any whitespace character like a blank character, tab, or line break
\S	Any character except for a blank character, tab, or line break
\t	Tab character
\uFFFF	Unicode character with the hexadecimal code FFFF. For example, the Euro symbol has the code 20AC

Element	Description
\v	Vertical tab (ASCII 11)
\w	Letter, digit, or underline
\W	Any character except for letters
\xnn	Particular character, where nn specifies the hexadecimal ASCII code
.*	Any number of any character (including no characters at all)

Table 13.8: Placeholders for characters

Quantifiers

Every pattern listed in **Table 13.8** represents exactly one instance of that kind. Using quantifiers, you can tell how many instances are parts of your pattern. For example, "\d{1,3}" represents a number occurring one to three times for a one-to-three digit number.

Element	Description
*	Preceding expression is not matched or matched once or several times (matches as much as possible)
*?	Preceding expression is not matched or matched once or several times (matches as little as possible)
.*	Any number of any character (including no characters at all)
?	Preceding expression is not matched or matched once (matches as much as possible)
??	Preceding expression is not matched or matched once (matches as little as possible)
{n,}	n or more matches
{n,m}	Inclusive matches between n and m
{n}	Exactly n matches
+	Preceding expression is matched once

Table 13.9: Quantifiers for patterns

Anchors

Anchors determine whether a pattern has to match a certain boundary. For example, the regular expression "\b\d{1,3}" finds numbers only up to three digits if these turn up separately in a string. The number "123" in the string "Bart123" would not qualify.

Element	Description
$	Matches at end of a string (\Z is less ambiguous for multi-line texts)
\A	Matches at beginning of a string, including multi-line texts
\b	Matches on word boundary (first or last characters in words)
\B	Must not match on word boundary
\Z	Must match at end of string, including multi-line texts
^	Must match at beginning of a string (\A is less ambiguous for multi-line texts)

Table 13.10: Anchor boundaries

Recognizing IP Addresses

Patterns such as an IP address can be very precisely described by regular expressions. Usually, you would use a combination of characters and quantifiers to specify which characters may occur in a string and how often:

```
$ip = "10.10.10.10"
$ip -match "\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b"
True
$ip = "a.10.10.10"
$ip -match "\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b"
False
$ip = "1000.10.10.10"
$ip -match "\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b"
False
```

The pattern is described here as four numbers (char: \d) between one and three digits (using the quantifier *{1,3}*) and anchored on word boundaries (using the anchor \b), meaning that it is surrounded by white space like blank characters, tabs, or line breaks. Checking is far from perfect since it is not verified whether the numbers really do lie in the permitted number range from 0 to 255.

```
# There still are entries incorrectly identified as valid IP addresses:
$ip = "300.400.500.999"
$ip -match "\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b"
True
```

Validating E-Mail Addresses

If you'd like to verify whether a user has given a valid e-mail address, use the following regular expression:

```
$email = "test@somewhere.com"
$email -match "\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b"
True
$email = ".@."
$email -match "\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b"
False
```

Whenever you look for an expression that occurs as a single "word" in text, delimit your regular expression by word boundaries (anchor: \b). The regular expression will then know you're interested only in those passages that are demarcated from the rest of the text by white space like blank characters, tabs, or line breaks.

The regular expression subsequently specifies which characters may be included in an e-mail address. Permissible characters are in square brackets and consist of "ranges" (for example, "A-Z0-9") and single characters (such as "._%+-"). The "+" behind the square brackets is a quantifier and means that at least one of the given characters must be present. However, you can also stipulate as many more characters as you wish.

Following this is "@" and, if you like, after it a text again having the same characters as those in front of "@". A dot (\.) in the e-mail address follows. This dot is introduced with a "\" character because the dot actually has a different meaning in regular expressions if it isn't within square brackets. The backslash ensures that the regular expression understands the dot behind it literally.

After the dot is the domain identifier, which may consist solely of letters ([A-Z]). A quantifier (*{2,4}*) again follows the square brackets. It specifies that the domain identifier may consist of at least two and at most four of the given characters.

However, this regular expression still has one flaw. While it does verify whether a valid e-mail address is in the text somewhere, there could be another text before or after it:

```
$email = "Email please to test@somewhere.com and reply!"
$email -match "\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b"
True
```

Because of "\b", when your regular expression searches for a pattern somewhere in the text, it only takes into account word boundaries. If you prefer to check whether the entire text corresponds to an authentic e-mail, use the elements for sentence beginnings (anchor: "^") and endings (anchor: "$") instead of word boundaries.

```
$email -match "^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$"
```

Simultaneous Searches for Different Terms

Sometimes search terms are ambiguous because there may be several ways to write them. You can use the "?" quantifier to mark parts of the search term as optional. In simple cases put a "?" after an optional character. Then the character in front of "?" may, but doesn't have to, turn up in the search term:

```
"color" -match "colou?r"
True
"colour" -match "colou?r"
True
```

Important

The "?" character here doesn't represent any character at all, as you might expect after using simple wildcards. For regular expressions, "?" is a quantifier and always specifies how often a character or expression in front of it may occur. In the example, therefore, "u?" ensures that the letter "u" may, but not necessarily, be in the specified location in the pattern. Other quantifiers are "*" (may also match more than one character) and "+" (must match characters at least once).

If you prefer to mark more than one character as optional, put the character in a sub-expression, which are placed in parentheses. The following example recognizes both the month designator "Nov" and "November":

```
"Nov" -match "\bNov(ember)?\b"
True
"November" -match "\bNov(ember)?\b"
True
```

If you'd rather use several alternative search terms, use the OR character "|":

```
"Bob and Ted" -match "Alice|Bob"
True
```

And if you want to mix alternative search terms with fixed text, use sub-expressions again:

```
# finds "and Bob":
"Peter and Bob" -match "and (Bob|Willy)"
True
```

```
# does not find "and Bob":
"Bob and Peter" -match "and (Bob|Willy)"
False
```

Case Sensitivity

In keeping with customary PowerShell practice, the *-match* operator is case insensitive. Use the operator *-cmatch* as alternative if you'd prefer case sensitivity:

```
# -match is case insensitive:
"hello" -match "heLLO"
True

# -cmatch is case sensitive:
"hello" -cmatch "heLLO"
False
```

If you want case sensitivity in only some pattern segments, use *–match*. Also, specify in your regular expression which text segments are case sensitive and which are insensitive. Anything following the "(?i)" construct is case insensitive. Conversely, anything following "(?-i)" is case sensitive. This explains why the word "test" in the below example is recognized only if its last two characters are lowercase, while case sensitivity has no importance for the first two characters:

```
"TEst" -match "(?i)te(?-i)st"
True
"TEST" -match "(?i)te(?-i)st"
False
```

If you use a .NET framework RegEx object instead of –match, it will work case-sensitive by default, much like –cmatch. If you prefer case insensitivity, either use the above construct to specify the option (i?) in your regular expression or submit extra options to the Matches() method (which is a lot more work):

```
 [regex]::matches("test", "TEST", "IgnoreCase")
```

Element	Description	Category
(xyz)	Sub-expression	
|	Alternation construct	Selection
\	When followed by a character, the character is not recognized as a formatting character but as a literal character	Escape
x?	Changes the x quantifier into a "lazy" quantifier	Option
(?xyz)	Activates of deactivates special modes, among others, case sensitivity	Option
x+	Turns the x quantifier into a "greedy" quantifier	Option
?:	Does not backtrack	Reference
?<name>	Specifies name for back references	Reference

Table 13.11: Regular expression elements

Of course, a regular expression can perform any number of detailed checks, such as verifying whether numbers in an IP address lie within the permissible range from 0 to 255. The problem is that this makes regular expressions long and hard to understand. Fortunately, you generally won't need to invest much time in learning complex regular expressions like the ones coming up. It's enough to know which regular expression to use for a particular pattern. Regular expressions for nearly all standard patterns can be downloaded from the Internet. In the following example, we'll look more closely at a complex regular expression that evidently is entirely made up of the conventional elements listed in **Table 13.11**:

```
$ip = "300.400.500.999"
$ip -match "\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b"
False
```

The expression validates only expressions running into word boundaries (the anchor is \b). The following sub-expression defines every single number:

```
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
```

The construct ?: is optional and enhances speed. After it come three alternatively permitted number formats separated by the alternation construct "|". *25[0-5]* is a number from *250* through *255. 2[0-4][0-9]* is a number from *200 through 249. Finally, [01]?[0-9][0-9]? is a number from 0-9 or 00-99 or 100-199. The quantifier "?" ensures that the preceding pattern must be included. The result is that the sub-expression describes numbers from 0 through 255. An IP address consists of four such numbers. A dot* always follows the first three numbers. For this reason, the following expression includes a definition of the number:

```
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
```

A dot, (\.), is appended to the number. This construct is supposed to be present three times ({3}). When the fourth number is also appended, the regular expression is complete. You have learned to create sub-expressions (by using parentheses) and how to iterate sub-expressions (by indicating the number of iterations in braces after the sub-expression), so you should now be able to shorten the first used IP address regular expression:

```
$ip = "10.10.10.10"
$ip -match "\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b"
True
$ip -match "\b(?:\d{1,3}\.){3}\d{1,3}\b"
True
```

Finding Information in Text

Regular expressions can recognize patterns. They can also filter data matching certain patterns from text. So, regular expressions are perfect for parsing raw data.

```
$rawtext = "If it interests you, my e-mail address is tobias@powershell.com."

# Simple pattern recognition:
$rawtext -match "\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b"
True

# Reading data matching the pattern from raw text:
$matches
Name                          Value
----                          -----
0                             tobias@powershell.com
$matches[0]
tobias@powershell.com
```

Does that also work for more than one e-mail addresses in text? Unfortunately, no. The *–match* operator finds only the first matching expression. So, if you want to find more than one occurrence of a pattern in raw text, you have to switch over to the *RegEx* object underlying the *–match* operator and use it directly.

Important

Since the RegEx object is case-sensitive by default, put the "(?i)" option before the regular expression to make it work like -match.

```
# A raw text contains several e-mail addresses. –match finds the first one only:
$rawtext = "test@test.com sent an e-mail that was forwarded to spam@junk.de."
$rawtext -match "\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b"
True
$matches
Name                Value
----                -----
0                  test@test.com

# A RegEx object can find any pattern but is case sensitive by default:
$regex = [regex]"(?i)\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b"
$regex.Matches($rawtext)
Groups  : {test@test.com}
Success : True
Captures : {test@test.com}
Index   : 4
Length   : 13
Value   : test@test.com

Groups  : {spam@junk.de}
Success : True
Captures : {spam@junk.de}
Index   : 42
Length  : 13
Value   : spam@junk.de

# Limit result to e-mail addresses:
$regex.Matches($rawtext) | Select-Object -Property Value
Value
-----
test@test.com
spam@junk.de

# Continue processing e-mail addresses:
$regex.Matches($rawtext) | ForEach-Object { "found: $($_.Value)" }
found: test@test.com
found: spam@junk.de
```

Searching for Several Keywords

You can use the alternation construct “|” to search for a group of keywords, and then find out which keyword was actually found in the string:

```
“Set a=1” -match “Get|GetValue|Set|SetValue”
True
$matches
Name                          Value
----                          -----
0                             Set
```

$matches tells you which keyword actually occurs in the string. But note the order of keywords in your regular expression—it’s crucial because the first matching keyword is the one selected. In this example, the result would be incorrect:

```
“SetValue a=1” -match “Get|GetValue|Set|SetValue”
True
$matches[0]
Set
```

Either change the order of keywords so that longer keywords are checked before shorter ones …:

```
“SetValue a=1” -match “GetValue|Get|SetValue|Set”
True
$matches[0]
SetValue
```

… or make sure that your regular expression is precisely formulated, and remember that you’re actually searching for single words. Insert word boundaries into your regular expression so that sequential order no longer plays a role:

```
“SetValue a=1” -match “\b(Get|GetValue|Set|SetValue)\b”
True
$matches[0]
SetValue
```

It’s true here, too, that -*match* finds only the first match. If your raw text has several occurrences of the keyword, use a *RegEx* object again:

```
$regex = [regex]”\b(Get|GetValue|Set|SetValue)\b”
$regex.Matches(“Set a=1; GetValue a; SetValue b=12”)
Groups   : {Set, Set}
Success  : True
Captures : {Set}
Index    : 0
Length   : 3
Value    : Set

Groups   : {GetValue, GetValue}
Success  : True
Captures : {GetValue}
Index    : 9
Length   : 8
Value    : GetValue
```

```
Groups   : {SetValue, SetValue}
Success  : True
Captures : {SetValue}
Index    : 21
Length   : 8
Value    : SetValue
```

Forming Groups

A raw text line is often a heaping trove of useful data. You can use parentheses to collect this data in sub-expressions so that it can be evaluated separately later. The basic principle is that all the data that you want to find in a pattern should be wrapped in parentheses because *$matches* will return the results of these sub-expressions as independent elements. For example, if a text line contains a date first, then text, and if both are separated by tabs, you could describe the pattern like this:

```
# Defining pattern: two characters separated by a tab
$pattern = "(.*)\t(.*)"

# Generate example line with tab character
$line = "12/01/2009`tDescription"

# Use regular expression to parse line:
$line -match $pattern
True

# Show result:
$matches
Name                              Value
----                             -----
2                                 Description
1                                 12/01/2009
0                                 12/01/2009     Description
$matches[1]
12/01/2009
$matches[2]
Description
```

When you use sub-expressions, $matches will contain the entire searched pattern in the first array element named "0". Sub-expressions defined in parentheses follow in additional elements. To make them easier to read and understand, you can assign sub-expressions their own names and later use the names to call results. To assign names to a sub-expression, type ? in parentheses for the first statement:

```
# Assign subexpressions their own names:
$pattern = "(?<Date>.*)\t(?<Text>.*)"

# Generate example line with tab character:
$line = "12/01/2009`tDescription"

# Use a regular expression to parse line:
$line -match $pattern
True
```

```
# Show result:
$matches
Name                           Value
----                           -----
Text                           Description
Date                           12/01/2009
0                              12/01/2009     Description
$matches.Date
12/01/2009
$matches.Text
Description
```

Each result retrieved by *$matches* for each sub-expression naturally requires storage space. If you don't need the results, discard them to increase the speed of your regular expression. To do so, type "?:" as the first statement in your sub-expression:

```
# Don't return a result for the second subexpression:
$pattern = "(?<Date>.*)\t(?:.*)"

# Generate example line with tab character:
$line = "12/01/2009`tDescription"

# Use a regular expression to parse line:
$line -match $pattern
True

# No more results will be returned for the second subexpression:
$matches
Name                           Value
----                           -----
Date                           12/01/2009
0                              12/01/2009     Description
```

Greedy or Lazy? Shortest or Longest Possible Result

Assume that you would like to evaluate month specifications in a logging file, but the months are not all specified in the same way. Sometimes you use the short form, other times the long form of the month name is used. As you've seen, that's no problem for regular expressions, because sub-expressions allow parts of a keyword to be declared optional:

```
"Feb" -match "Feb(ruary)?"
True
$matches[0]
Feb
"February" -match "Feb(ruary)?"
True
$matches[0]
February
```

In both cases, the regular expression recognizes the month, but returns different results in *$matches*. By default, the regular expression is “greedy” and returns the longest possible match. If the text is “February,” then the expression will search for a match starting with “Feb” and then continue searching “greedily” to check whether even more characters match the pattern. If they do, the entire (detailed) text is reported back: February.

If your main concern is just standardizing the names of months, you would probably prefer getting back the shortest possible text: Feb. To switch regular expressions to work lazy (returning the shortest possible match), add “?” to the expression. “Feb(ruary)??” now stands for a pattern that starts with “Feb”, followed by zero or one occurance of “ruary” (Quantifier “?”), and returning only the shortest possible match (which is turned on by the second “?”).

```
"Feb" -match "Feb(ruary)??"
True
$matches[0]
Feb
"February" -match "Feb(ruary)??"
True
$matches[0]
February
```

Finding String Segments

Our last example, which locates text segments, shows how you can use the elements listed in Table 13.11 to easily gather surprising search results. If you type two words, the regular expression will retrieve the text segment between the two words if at least one word is, and not more than six other words are, in between the two words. This example shows how complex (and powerful) regular expressions can get. If you think that’s cool, you should grab yourself a book on regular expressions and dive deeper:

```
"Find word segments from start to end" -match "\bstart\W+(?:\w+\W+){1,6}?end\b"
True
$matches[0]
Name                           Value
----                           -----
0                              start to end
```

Replacing a String

You already know how to replace a string because you know the string –replace operator. Simply tell the operator what term you want to replace in a string:

```
"Hello, Ralph" -replace "Ralph", "Martina"
Hello, Martina
```

But simple replacement isn’t always sufficient, so you can also use regular expressions for replacements. Some of the following examples show how that could be useful.

Let’s say you’d like to replace several different terms in a string with one other term. Without regular expressions, you’d have to replace each term separately. With regular expressions, simply use the alternation operator, “|”:

```
"Mr. Miller and Mrs. Meyer" -replace "(Mr.|Mrs.)", "Our client"
Our client Miller and Our client Meyer
```

You can type any term in parentheses and use the "|" symbol to separate them. All the terms will be replaced with the replacement string you specify.

Using Back References

This last example replaces specified keywords anywhere in a string. Often, that's sufficient, but sometimes you don't want to replace a keyword everywhere it occurs but only when it occurs in a certain context. In such cases, the context must be defined in some way in the pattern. How could you change the regular expression so that it replaces only the names Miller and Meyer? Like this:

```
"Mr. Miller, Mrs. Meyer and Mr. Werner" -replace "(Mr.|Mrs.)\s*(Miller|Meyer)", "Our client"
Our client, Our client and Mr. Werner
```

The back references don't seem to work. Can you see why? "$1" and "$2" look like PowerShell variables, but in reality they are part of the regular expression. As a result, if you put the replacement string inside double quotes, PowerShell replaces "$2" with the PowerShell variable $2, which is probably undefined. Use single quotation marks instead, or add a backtick to the "$" special character so that PowerShell won't recognize it as its own variable and replace it:

```
# Replacement text must be inside single quotation marks so that the PS variable $2:
"Mr. Miller, Mrs. Meyer and Mr. Werner" -replace "(Mr.|Mrs.)\s*(Miller|Meyer)", 'Our client $2'
Our client Miller, Our client Meyer and Mr. Werner

# Alternatively, $ can also be masked by `$:
"Mr. Miller, Mrs. Meyer and Mr. Werner" -replace "(Mr.|Mrs.)\s*(Miller|Meyer)", "Our client `$2"
Our client Miller, Our client Meyer and Mr. Werner
```

Putting Characters First at Line Beginnings

Replacements can also be made in multiple instances in text of several lines. For example, when you respond to an e-mail, usually the text of the old e-mail is quoted in your new e-mail and marked with ">" at the beginning of each line. Regular expressions can do the marking.

However, to accomplish this, you need to know a little more about "multi-line" mode. Normally, this mode is turned off, and the "^" anchor represents the text beginning and the "$" the text ending. So that these two anchors refer respectively to the line beginning and line ending of a text of several lines, the multi-line mode must be turned on with the "(?m)" statement. Only then will *–replace* substitute the pattern in every single line. Once the multi-line mode is turned on, the anchors "^" and "\A", as well as "$" and "\Z", will suddenly behave differently. "\A" will continue to indicate the text beginning, while "^" will mark the line ending; "\Z" will indicate the text ending, while "$" will mark the line ending.

```
# Using Here-String to create a text of several lines:
$text = @"
>> Here is a little text.
>> I want to attach this text to an e-mail as a quote.
>> That's why I would put a ">" before every line.
>> "@
>>
$text
Here is a little text.
I want to attach this text to an e-mail as a quote.
That's why I would put a ">" before every line.
```

```
# Normally, -replace doesn't work in multiline mode. For this reason,
# only the first line is replaced:
$text -replace "^", "> "
> Here is a little text.
I want to attach this text to an e-mail as a quote.
That's why I would put a ">" before every line.

# If you turn on multiline mode, replacement will work in every line:
$text -replace "(?m)^", "> "
> Here is a little text.
> I want to attach this text to an e-mail as a quote.
> That's why I would put a ">" before every line.

# The same can also be accomplished by using a RegEx object,
# where the multiline option must be specified:
[regex]::Replace($text, "^", "> ", [Text.RegularExpressions.RegExOptions]::Multiline)
> Here is a little text.
> I want to attach this text to an e-mail as a quote.
> That's why I would put a ">" before every line.

# In multiline mode, \A stands for the text beginning and ^ for the line beginning:
[regex]::Replace($text, "\A", "> ", [Text.RegularExpressions.RegExOptions]::Multiline)
> Here is a little text.
I want to attach this text to an e-mail as a quote.
That's why I would put a ">" before every line.
```

Removing White Space

Regular expressions can perform routine tasks as well, such as remove superfluous white space. The pattern describes a blank character (char: "\s") that occurs at least twice (quantifier: "{2,}"). That is replaced with a normal blank character.

```
"Too    many    blank    characters" -replace "\s{2,}", " "
Too many blank characters
```

Finding and Removing Doubled Words

How is it possible to find and remove doubled words in text? Here, you can use back referencing again. The pattern could be described as follows:

```
"\b(\w+)(\s+\1){1,}\b"
```

The pattern searched for is a word (anchor: "\b"). It consists of one word (the character "\w" and quantifier "+"). A blank character follows (the character "\s" and quantifier "?"). This pattern, the blank character and the repeated word, must occur at least once (at least one and any number of iterations of the word, quantifier "{1,}"). The entire pattern is then replaced with the first back reference, that is, the first located word.

```
# Find and remove doubled words in a text:
"This this this is a test" -replace "\b(\w+)(\s+\1){1,}\b", '$1'
This is a test
```

Summary

Text is defined either by single or double quotation marks. If you use double quotation marks, PowerShell will replace PowerShell variables and special characters in the text. Text enclosed in single quotation marks remains as-is. If you want to prompt the user for input text, use the Read-Host cmdlet. Multi-line text can be defined with Here-Strings, which start with @"**(Enter)** and end with "@ **(Enter)**.

By using the format operator –f, you can compose formatted text. This gives you the option to display text in different ways or to set fixed widths to output text in aligned columns (**Table 13.3** through Table **13.5**). Along with the formatting operator, PowerShell has a number of string operators you can use to validate patterns or to replace a string (**Table 13.2**).

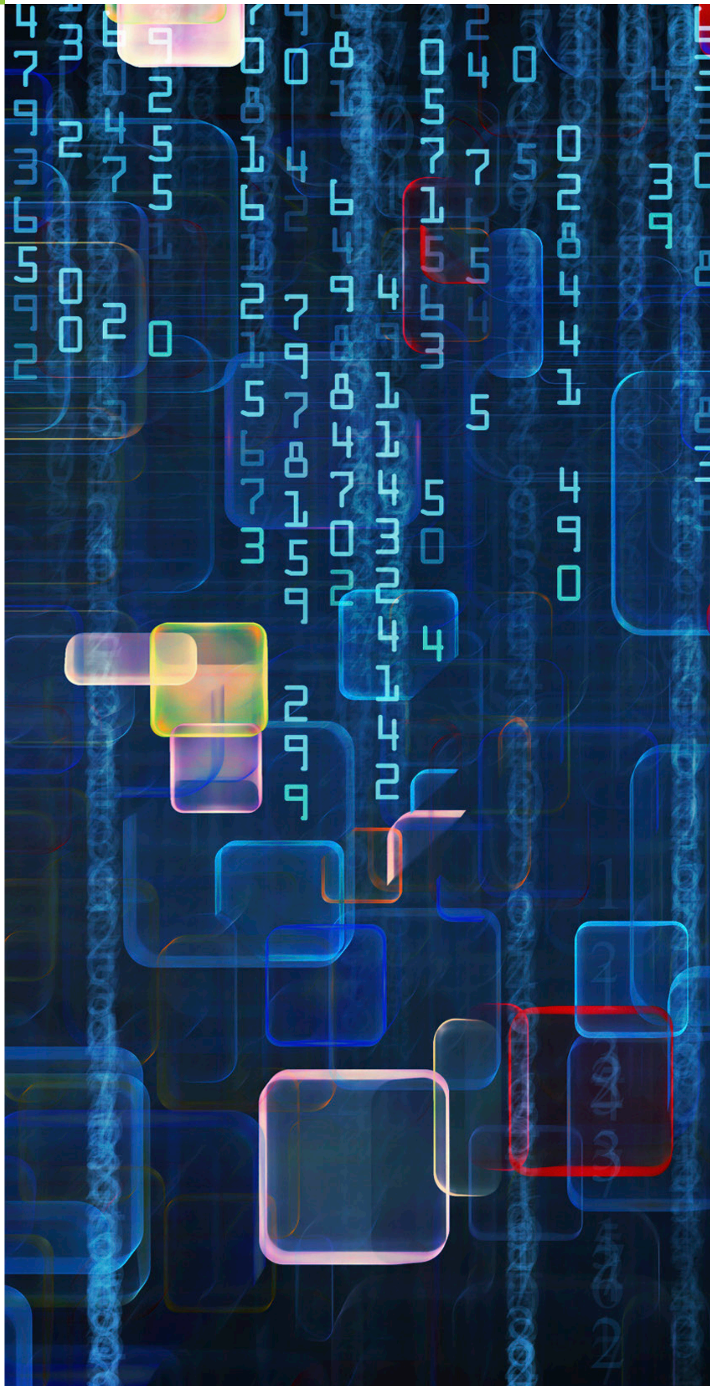
PowerShell stores text in string objects, which support methods to work on the stored text. You can use these methods by typing a dot after the string object (or the variable in which the text is stored) and then activating auto complete (**Table 13.6**). Along with the dynamic methods that always refer to text stored in a string object, there are also static methods that are provided directly by the string data type by qualifying the string object with "[string]::".

The simplest way to describe patterns is to use the simple wildcards in **Table 13.7**. Simple wildcard patterns, while easy to use, only support very basic pattern recognition. Also, simple wildcard patterns can only recognize the patterns; they cannot extract data from them.

A far more sophisticated tool are regular expressions. They consist of very specific placeholders, quantifiers and anchors listed in Table **13.11**. Regular expressions precisely identify even complex patterns and can be used with the operators *-match* or *–replace*. Use the .NET object [regex] if you want to match multiple pattern instances.

Chapter 14. Conditions

In today's world, data is no longer presented in plain-text files. Instead, XML (Extensible Markup Language) has evolved to become a de facto standard because it allows data to be stored in a flexible yet standard way. PowerShell takes this into account and makes working with XML data much easier than before.



Topcs Covered:

Taking a Look At XML Structure

XML uses tags to uniquely identify pieces of information. A tag is a pair of angle brackets like the ones used for HTML documents. Typically, a piece of information is delimited by a start and end tag. The end tag is preceded by "/"; the result is called a "node", and in the next example, the node is called "Name":

```
<Name>Tobias Weltner</Name>
```

Nodes can be decorated with attributes. Attributes are stored in the start tag of the node like this:

```
<staff branch="Hanover" Type="sales">...</staff>
```

If a node has no particular content, its start and end tags can be combined, and the ending symbol "/" drifts toward the end of the tag. If the branch office in Hanover doesn't have any staff currently working in the field, the tag could look like this:

```
<staff branch="Hanover" Type="sales"/>
```

The following XML structure describes two staff members of the Hanover branch office who are working in the sales department.

```
<staff branch="Hanover" Type="sales">
  <employee>
    <Name>Tobias Weltner</Name>
    <function>management</function>
    <age>39</age>
  </employee>
  <employee>
    <Name>Cofi Heidecke</Name>
    <function>security</function>
    <age>4</age>
  </employee>
</staff>
```

The XML data is wrapped in an XML node which is the top node of the document:

```
<?xml version="1.0" ?>
```

This particular header contains a version attribute which declares that the XML structure conforms to the specifications of XML version 1.0. There can be additional attributes in the XML header. Often you find a reference to a "schema", which is a formal description of the structure of that XML file. The schema could, for example, specify that there must always be a node called "staff" as part of staff information, which in turn could include as many sub-nodes named "staff" as required. The schema would also specify that information relating to name and function must also be defined for each staff member.

Because XML files consist of plain text, you can easily create them using any editor or directly from within PowerShell. Let's save the previous staff list as an xml file:

```
$xml = @'
<?xml version="1.0" standalone="yes"?>
<staff branch="Hanover" Type="sales">
  <employee>
    <Name>Tobias Weltner</Name>
    <function>management</function>
    <age>39</age>
  </employee>
  <employee>
    <Name>Cofi Heidecke</Name>
    <function>security</function>
    <age>4</age>
  </employee>
</staff>
'@ | Out-File $env:temp\employee.xml
```

Note

XML is case-sensitive!

Loading and Processing XML Files

To read and evaluate XML, you can either convert the text to the XML data type, or you can instantiate a blank XML object and load the XML from a file or a URL in the Internet. This line would read the content from a file $env:temp\employee.xml and convert it to XML:

```
$xmldata = [xml](Get-Content $env:temp\employee.xml)
```

A faster approach uses a blank XML object and its Load() method:

```
$xmldata = New-Object XML
$xmldata.Load("$env:temp\employee.xml")
```

Conversion or loading XML from a file of course only works when the XML is valid and contains no syntactic errors. Else, the conversion will throw an exception.

Once the XML data is stored in an XML object, it is easy to read its content because PowerShell automatically turns XML nodes

```
$xmldata.staff.employee

Name                                  function                              Age
----                                 -----                                -----
Tobias Weltner                        management                             39
Cofi Heidecke                         security                                4
```

Accessing Single Nodes and Modifying Data

To pick out a specific node from a set of nodes, you can use the PowerShell pipeline and Where-Object. This would pick out a particular employee from the list of staff. As you will see, you can not only read data but also change it.

```
$xmldata.staff.employee | Where-Object { $_.Name -match "Tobias Weltner" }
Name                                        function                                    Age
----                                        -----                                       -----
Tobias Weltner                              management                                   39

$employee = $xmldata.staff.employee | Where-Object { $_.Name -match "Tobias Weltner" }
$employee.function = "vacation"
$xmldata.staff.employee
Name                                        function                                    Age
----                                        -----                                       -----
Tobias Weltner                              vacation                                    39
```

If you want to save changes you applied to XML data, call the Save() method:

```
$xmldata.Save("$env:temp\updateddata.xml")
```

Using SelectNodes() to Choose Nodes

Another way of picking nodes is to use the method SelectNode() and its so-called XPath query language. So, to get to the employee data below the staff node, use this approach:

```
$xmldata.SelectNodes('staff/employee')

Name                                     function                                 Age
----                                     -----                                    -----
Tobias Weltner                           management                                39
Cofi Heidecke                            security                                   4
```

The result is pretty much the same as before, but XPath is very flexible and supports wildcards and additional control. The next statement retrieves just the first employee node:

```
Name                                           function                                         Age
----                                          -----                                            -----
Tobias Weltner                                management                                         39
```

If you'd like, you can get a list of all employees who are under the age of 18:

```
$xmldata.SelectNodes('staff/employee[last()]')
$xmldata.SelectNodes('staff/employee[position()>1]')
```

Tip

Alternatively, you can also use an XpathNavigator:

```
# Create navigator for XML:
$xpath = [System.XML.XPath.XPathDocument][System.IO.TextReader][System.IO.StringReader]`
(Get-Content $env:temp\employee.xml | Out-String)
$navigator = $xpath.CreateNavigator()

# Output the last employee name of the Hanover branch office:
$query = "/staff[@branch='Hanover']/employee[last()]/Name"
$navigator.Select($query) | Format-Table Value
Value
-----
Cofi Heidecke

# Output all employees of the Hanover branch office except for Tobias Weltner:
$query = "/staff[@branch='Hanover']/employee[Name!='Tobias Weltner']"
$navigator.Select($query) | Format-Table Value
Value
-----
Cofi Heidecke
```

Accessing Attributes

Attributes are pieces of information that describe an XML node. If you'd like to read the attributes of a node, use *Attributes:*

```
$xmldata.staff.Attributes
#text
-----
Hanover
sales
```

Use GetAttribute() if you'd like to query a particular attribute:

```
$xmldata.staff.GetAttribute("branch")
Hanover
```

Use SetAttribute() to specify new attributes or modify (overwrite) existing ones:

```
$xmldata.staff.SetAttribute("branch", "New York")
$xmldata.staff.GetAttribute("branch")
New York
```

Adding New Nodes

If you'd like to add new employees to your XML, use CreateElement() to create an employee element and then fill in the data. Finally, add the element to the XML:

```
# Create new node:
$newemployee = $xmldata.CreateElement("employee")
$newemployee.InnerXML = '<Name>Bernd Seiler</Name><function>expert</function>'

# Write nodes in XML:
$xmldata.staff.AppendChild($newemployee)

# Check result:
$xmldata.staff.employee

Name                                    function                                   Age
----                                     -----                                     -----
Tobias Weltner                          management                                  39
Cofi Heidecke                           security                                    4
Bernd Seiler                            expert

# Output plain text:
$xmldata.get_InnerXml()
<?xml version="1.0"?><Branch office staff="Hanover" Type="sales"><employee>
<Name>Tobias Weltner</Name><function>management</function><age>39</age>
</employee><employee><Name>Cofi Heidecke</Name><function>security</function>
<age>4</age></employee><employee><Name>Bernd Seiler</Name><function>
expert</function></employee></staff>
```

Exploring the Extended Type System

The PowerShell Extended Type System (ETS) is XML-based, too. The ETS is responsible for turning objects into readable text. PowerShell comes with a set of xml files that all carry the extension ".ps1xml". There are format-files and type-files. Format-files control which object properties are shown and how the object structure is represented. Type-format files control which additional properties and methods should be added to objects.

With the basic knowledge about XML that you gained so far, you can start exploring the ETS XML files and learn more about the inner workings of PowerShell.

The XML Data of the Extended Type System

Whenever PowerShell needs to convert an object into text, it searches through its internal "database" to find information about how to best format and display the object. This database really is a collection of XML files in the PowerShell root folder *$pshome:*

```
Dir $pshome\*.format.ps1xml
```

All these files define a multitude of Views, which you can examine using PowerShell XML support.

```
[xml]$file = Get-Content "$pshome\dotnettypes.format.ps1xml"
$file.Configuration.ViewDefinitions.View
Name                                  ViewSelectedBy                        TableControl
----                                  --------------                        ------------
System.Reflection.Assembly            ViewSelectedBy                        TableControl
System.Reflection.AssemblyName        ViewSelectedBy                        TableControl
System.Globalization.CultureInfo      ViewSelectedBy                        TableControl
System.Diagnostics.FileVersionInfo    ViewSelectedBy                        TableControl
System.Diagnostics.EventLogEntry      ViewSelectedBy                        TableControl
System.Diagnostics.EventLog           ViewSelectedBy                        TableControl
System.Version                        ViewSelectedBy                        TableControl
System.Drawing.Printing.PrintDo...    ViewSelectedBy                        TableControl
Dictionary                            ViewSelectedBy                        TableControl
ProcessModule                         ViewSelectedBy                        TableControl
process                               ViewSelectedBy                        TableControl
PSSnapInInfo                          ViewSelectedBy
PSSnapInInfo                          ViewSelectedBy                        TableControl
Priority                              ViewSelectedBy                        TableControl
StartTime                             ViewSelectedBy                        TableControl
service                               ViewSelectedBy                        TableControl
(...)
```

Finding Pre-Defined Views

Pre-defined views are interesting because you can use the *-View* parameter to change the way PowerShell presents results with the cmdlets *Format-Table* or *Format-List*.

```
Get-Process | Format-Table -View Priority
Get-Process | Format-Table -View StartTime
```

To find out which views exist, take a look into the format.ps1xml files that describe the object type.

```
[xml]$file = Get-Content "$pshome\dotnettypes.format.ps1xml"
$view = @{ Name='ObjectType'; Expression= {$_.ViewSelectedBy.TypeName}}
$file.Configuration.ViewDefinitions.View | Select-Object Name, $view |
Where-Object { $_.Name -ne $_. ObjectType } | Sort-Object ObjectType
```

```
Name                                    ObjectType
----                                   ----------
Dictionary                              System.Collections.DictionaryEntry
DateTime                                System.DateTime
Priority                                System.Diagnostics.Process
StartTime                               System.Diagnostics.Process
process                                 System.Diagnostics.Process
process                                 System.Diagnostics.Process
ProcessModule                           System.Diagnostics.ProcessModule
DirectoryEntry                          System.DirectoryServices.DirectoryEntry
PSSnapInInfo                            System.Management.Automation.PSSnapI...
PSSnapInInfo                            System.Management.Automation.PSSnapI...
service                                 System.ServiceProcess.ServiceController
```

Here you see all views defined in this XML file. The object types for which the views are defined are listed in the second column. The *Priority* and *StartTime* views, which we just used, are on that list. However, the list just shows views that use Table format. To get a complete list of all views, here is a more sophisticated example:

```
Name                          ObjectType                   Type
----                         ----------                   ----
Dictionary                   System.Collections.Dict... Table
System.Collections.Dict... System.Collections.Dict... List
System.Diagnostics.Even... System.Diagnostics.Even... List
System.Diagnostics.Even... System.Diagnostics.Even... Table
System.Diagnostics.Even... System.Diagnostics.Even... Table
System.Diagnostics.Even... System.Diagnostics.Even... List
System.Diagnostics.File... System.Diagnostics.File... List
System.Diagnostics.File... System.Diagnostics.File... Table
Priority                     System.Diagnostics.Process Table
process                       System.Diagnostics.Process Wide
StartTime                     System.Diagnostics.Process Table
process                       System.Diagnostics.Process Table
PSSnapInInfo                  System.Management.Autom... Table
PSSnapInInfo                  System.Management.Autom... List
System.Reflection.Assembly System.Reflection.Assembly Table
System.Reflection.Assembly System.Reflection.Assembly List
System.Security.AccessC... System.Security.AccessC... List
System.Security.AccessC... System.Security.AccessC... Table
service                       System.ServiceProcess.S... Table
System.ServiceProcess.S... System.ServiceProcess.S... List
System.TimeSpan              System.TimeSpan             Wide
System.TimeSpan              System.TimeSpan             Table
System.TimeSpan              System.TimeSpan             List
```

Remember there are many format.ps1xml-files containing formatting information. You'll only get a complete list of all view definitions when you generate a list for all of these files.