**10.6.1 Adding a Property**

Now let’s add a new member to this object. You could use Add-Member (and typically you would), but we’re talking about the plumbing here, so we’ll do it the hard way. First, you need to create the NoteProperty object that you want to add. Do this with the New-Object cmdlet:

PS> $np = New-Object `

-TypeName System.Management.Automation.PSNoteProperty `

-ArgumentList hi, 'Hello there'

Next, add it to the member collection:

PS> $f.PSObject.Members.add($np)

and you’re finished (so it wasn’t that hard after all). The hi member has been added to this object, so try it out:

PS> $f.hi

Hello there

All of the normal members are still there. Now look at the member in the member collection:

PS> $f.PSObject.Members | where {$\_.name -match '^hi'}

MemberType : NoteProperty

IsSettable : True

IsGettable : True

Value : Hello there

TypeNameOfValue : System.String

Name : hi

IsInstance : True

Notice the Value member on the object. Because you can get at the member, you can also set the member

PS> ($f.PSObject.Members | where {

$\_.name -match '^hi'}).value = 'Goodbye!'

PS> $f.hi

Goodbye!

which is equivalent to setting the property directly on $f:

PS> $f.hi = 'Hello again!'

PS> $f.PSObject.Members | where {$\_.name -match '^hi'}

MemberType : NoteProperty

IsSettable : True

IsGettable : True

Value : Hello again!

TypeNameOfValue : System.String

Name : hi

IsInstance : True

The Value member on the note property is Hello again!

In [section 10.4.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-104-86.xhtml#ch10lev2sec11) you saw a different type of note property used when constructing objects out of modules. This type of note property is backed by a variable. You can also create an instance of this type of property. But first you need a variable to use to back the property value:

PS> [int] $VariableProperty = 0

Now create the PSVariableProperty object, passing in the variable to bind:

PS> $vp = New-Object `

-TypeName System.Management.Automation.PSVariableProperty `

-ArgumentList (Get-Variable VariableProperty)

Note that the name of the property and the name of the variable will be the same. Add the property

PS> $f.psobject.members.add($vp)

and verify that it can be read and written:

PS> $f.VariableProperty

0

PS> $f.VariableProperty = 7

PS> $f.VariableProperty

7

You can read and write integers, but the backing variable was constrained to be an integer. Let’s verify that the constraint was preserved by trying to assign a string to it:

PS> $f.VariableProperty = 'Hi'

Cannot convert value "Hi" to type "System.Int32".

Error: "Input string was not in a correct format."

At line:1 char:1

+ $f.VariableProperty = 'Hi'

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : MetadataError: (:) [], ArgumentTransformationMetadataException

+ FullyQualifiedErrorId : RuntimeException

You get the error like you saw in [section 10.4.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-104-86.xhtml#ch10lev2sec11) when you exported a constrained variable from a module as a property.

**10.6.2 Shadowing an Existing Property**

There’s one last item to cover in our discussion of the plumbing: the mechanism that allows you to bypass the adapted members and lets you get at the raw object underneath. This is accomplished through another special member on PSObject called PSBase. This member allows you to get at the object directly, bypassing all the synthetic member lookup. It also makes it possible to create a synthetic member to adapt an existing member. We can clarify this with an example. Say you want to change the name property on a DirectoryInfo object to always return the name in uppercase. Here’s what it looks like unadapted:

PS> $f = Get-Item c:\windows

PS> $f.name

windows

To do this, create a new PSProperty object called Name that will shadow the existing property:

PS> $n=New-Object -TypeName Management.Automation.PSScriptProperty `

-ArgumentList name,{$this.psbase.name.ToUpper()}

In the body of the scriptblock for this PSProperty, you’ll use $this.psbase to get at the name property on the base object (if you accessed the name property directly, you’d be calling yourself). You apply the ToUpper() method on the string returned by name to acquire the desired result. Now add the member to the object’s member collection

PS> $f.psobject.members.add($n)

and try it out:

PS> $f.name

WINDOWS

When you access the name property on this object, the synthetic member you created gets called instead of the base member, so the name is returned in uppercase. The base object’s name property is unchanged and can be retrieved through psbase.name:

PS> $f.psbase.name

windows

Although this isn’t a technique that you’ll typically use on a regular basis, it allows you to do some pretty sophisticated work. You could use it to add validation logic, for example, and prevent a property from being set to an undesired value. You could also use it to log accesses to a property to gather information about how your script or application is being used.

With a solid understanding of the plumbing, you’re ready to use everything you’ve learned and do some applied metaprogramming. In the next section, you’ll learn how to write a domain-specific extension to PowerShell.

## 10.7 Extending the PowerShell Language

In the previous section, you learned how to add members to existing objects one at a time, but sometimes you’ll want to construct new types rather than extend the existing types. In this section, we’ll explain how to do that and also how to use scripting techniques to add the ability to create objects to the PowerShell language.

### 10.7.1 Little Languages

The idea of little languages—small, domain-specific languages—has been around for a long time. This was one of the powerful ideas that made the UNIX environment so attractive. Many of the tools that were the roots for today’s dynamic languages came from this environment.

In effect, all programs are exercises in building their own languages. You create the nouns (objects) and verbs (methods or functions) in this language. These patterns are true for all languages that support data abstraction. Dynamic languages go further because they allow you to extend how the nouns, verbs, and modifiers are composed in the language. For example, in a language such as C#, it would be difficult to add a new looping construct. In PowerShell, this is minor. To illustrate how easy it is, let’s define a new looping keyword called loop. This construct will repeat the body of the loop for the number of times the first argument specifies. You can add this keyword by defining a function that takes a number and scriptblock. Here’s the definition:

PS> function loop ([int] $i, [scriptblock] $b) {

while ($i-- -gt 0) { . $b }

}

Try it out:

PS> loop 3 { 'Hello World' }

Hello world

Hello world

Hello world

In a few lines of code, you’ve added a new flow-control statement to the PowerShell language that looks pretty much like any of the existing flow-control statements. The only problem is that the opening brace has to be on the same line as the command—otherwise, it will be treated as two statements.

#### Note

As of version 4, PowerShell does have a way to create real language extensions where everything doesn’t have to be on the same line. This is how the language extensions for PowerShell DSC were implemented. Unfortunately, these capabilities are only exposed in the form of rather hard-to-use APIs. Although there are a number of community examples demonstrating how to use these APIs, we’re not going to cover them in this book because they’re still subject to change. (Microsoft may change the APIs as part of the process of properly exposing them.)

Now let’s change gears a bit to talk more about types.

### 10.7.2 Type Extension

You might have noticed that all the examples we’ve looked at so far involve adding members to instances. But what about adding members to types? Having to explicitly add members to every object you encounter would be pretty tedious, no matter how clever you are.

#### Note

Nope—still not talking about classes. Wait for [chapter 19](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-19-143.xhtml#ch19). Patience is a virtue (or so they tell us).

You need some way to extend types instead of individual instances. As you might expect, PowerShell lets you do exactly this. In the following sections, we’ll introduce the mechanisms that PowerShell provides which let you extend types.

Type extension is performed in PowerShell through a set of XML configuration files. These files are usually loaded at startup time, but they can be extended after the shell has started. In this section, you’ll learn how to take advantage of these features.

Let’s look at an example. Consider an array of numbers. It’s fairly common to sum up a collection of numbers; unfortunately, there’s no Sum() method on the Array class:

PS> (1,2,3,4).Sum()

Method invocation failed because [System.Int32] does not contain a method named 'Sum'.

At line:1 char:1

+ (1,2,3,4).sum()

+ ~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidOperation: (:) [], RuntimeException

+ FullyQualifiedErrorId : MethodNotFound

Using the techniques, we’ve discussed, you could add such a method to this array:

PS> $a = (1,2,3,4)

PS> $a = Add-Member -PassThru -in $a scriptmethod sum {

$r=0

foreach ($e in $this) {$r += $e}

$r

}

and finally use it:

PS> $a.sum()

10

But this would be painful to do for every instance of an array. What you need is a way to attach new members to a type, rather than through an instance. PowerShell does this through type configuration files. These configuration files are stored in the installation directory for PowerShell and loaded at startup. The installation directory path for PowerShell is stored in the $PSHome variable, so it’s easy to find these files. They have the word type in their names and have an extension of .ps1xml:

PS> Get-ChildItem $pshome/\*type\*.ps1xml

You don’t want to update the default installed types files because when you install updates for PowerShell, they’ll likely be overwritten and your changes will be lost. Instead, create your own custom types file containing the specification of the new member for System.Array. Once you’ve created the file, you can use the Update -TypeData cmdlet to load it. The definition for the Sum() method extension you want to add to System.Array is shown next.

#### Listing 10.6: Type file for Sum() method extension

<Types>

<Type>

<Name>System.Array</Name>

<Members>

<ScriptMethod>

<Name>Sum</Name>

<Script>

$r=$null

foreach ($e in $this) {$r += $e}

$r

</Script>

</ScriptMethod>

</Members>

</Type>

</Types>

This definition is saved to a file called SumMethod.ps1xml. Now load the file and update the type system definitions:

PS> Update-TypeData SumMethod.ps1xml

If the file loads successfully, you won’t see any output. You can now try out the Sum() function:

PS> (1,2,3,4,5).Sum()

15

It works. And because of the way the script was written, it will work on any type that can be added. Let’s add strings:

PS> ('abc','def','ghi').Sum()

abcdefghi

You can even use it to add hashtables:

PS> (@{a=1},@{b=2},@{c=3}).Sum()

Name Value

---- -----

c 3

a 1

b 2

You can see that the result is the composition of all three of the original hashtables. You can even use it to put a string back together:

PS> ([char[]] 'hal' | foreach{[char]([int]$\_+1)}).Sum()

ibm

Here you break the original string into an array of characters, add 1 to each character, and then use the Sum() method to add them all back into a string.

You should take time to examine the set of type configuration files that are part of the default PowerShell installation. Examining these files is a good way to see what you can accomplish using these tools.

#### Warning

Make sure you don’t make any changes to these files because bad things will happen to your PowerShell implementation.

Starting with PowerShell v3 you can modify types dynamically in a script. Instead of creating a type data file, you can imperatively extend a type with Update-TypeData. The dynamically modified type data is available only in the session in which you apply it—exactly the same as if you used a types file. You can also remove modified type data (dynamic or from a type file) using Remove-TypeData.

Let’s use the example from this section, but this time we’ll perform the update dynamically. Save this listing as dynamictypes.ps1.

#### Listing 10.7: Updating type data dynamically

Update-TypeData -TypeName System.Array -MemberName Sum `

-MemberType ScriptMethod -Value {

$r=$null

foreach ($e in $this) {$r += $e}

$r

} -Force 1

"`nSum array of numbers:" 2

(1,2,3,4,5).Sum()

"`nSum array of strings:"

("abc","def","ghi").Sum()

"`nSum array of hashtables:"

(@{a=1},@{b=2},@{c=3}).Sum()

"`nPut string back together:"

([char[]] "hal" | foreach{[char]([int]$\_+1)}).Sum()

Remove-TypeData -TypeName System.Array 3

"`nSum array of numbers:"

(1,2,3,4,5).Sum()

* 1 Defining dynamic type data
* 2 Summing an array
* 3 Removing dynamic type data

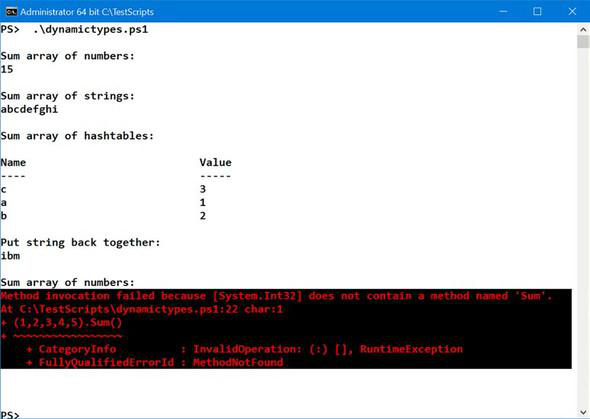
When defining dynamic types 1 you need to supply Update-TypeData with several pieces of information:

* Type to be modified
* Name of the new member
* Type of the new member
* Value or code used to define the new member

If you compare [listing 10.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-107-89.xhtml#ch10ex07) with [listing 10.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-107-89.xhtml#ch10ex06), you’ll see the very same data in both. When you modify types dynamically, you should use the -Force parameter. This doesn’t turn you into a Jedi knight but it ensures that the modification of the type will be applied even if you’ve already performed the action. It’s useful when developing and testing or if you need to rerun the script in the same session multiple times.

The same summations 2 are perfomed as in the previous discussion on using type files—namely, summing arrays of numbers, strings, and hashtables and putting a string back together. The results are shown in [figure 10.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-107-89.xhtml#ch10fig05).

Figure 10.5: Running a script that dynamically updates types



You can remove the modified type data 3 using Remove-TypeData. You’ll remove all type modifications for that particular type. If you then try to use the Sum() method on an array, you’ll receive a Method Invocation error because the method doesn’t exist. As a further exercise in dynamically modifying types, we’ll leave it to you to add a method to System.String to reverse the string. (Hint: see the ScriptMethod in [section 10.2.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-102-84.xhtml#ch10lev2sec7).)

We’ve covered an enormous amount of material so far in this chapter, introducing ideas that are new to a lot of users. If you’ve hung on to this point, congratulations! There are only a few more topics to complete your knowledge of metaprogramming with PowerShell. Scriptblocks, dynamic modules, and closures can be passed around, invoked, and assigned at runtime, but the body of these blocks is still defined at compile time. In the next section we’ll expand our repertoire of techniques by looking at ways to dynamically create code.

## 10.8 Building Script Code at Runtime

This final section presents the mechanisms that PowerShell provides for compiling script code and creating scriptblocks at runtime. To say that you’re compiling when PowerShell is an interpreted language may sound odd, but that’s what creating a scriptblock is: a piece of script text is compiled into an executable object. In addition, PowerShell provides mechanisms for directly executing a string, bypassing the need to first build a scriptblock. In the next few sections we’ll look at how each of these features works.

### 10.8.1 The Invoke-Expression Cmdlet

The Invoke-Expression cmdlet is a way to execute an arbitrary string as a piece of code. It takes the string, compiles it, and then immediately executes it in the current scope. Here’s an example:

PS> Invoke-Expression -Command '$a=2+2; $a'

4

In this example, the script passed to the cmdlet assigned the result of 2+2 to $a and wrote $a to the output stream. Because this expression was evaluated in the current context, it should also have affected the value of $a in the global scope:

PS> $a

4

You see that it did. Now invoke another expression:

PS> Invoke-Expression '$a++'

PS> $a

5

Evaluating this expression changes the value of $a to 5. There are no limits on what you can evaluate with Invoke-Expression. It can take any arbitrary piece of script code.

#### Warning

Danger! Danger! Danger! If you ever find yourself using the Invoke-Expression cmdlet (or the corresponding APIs) in production code, you’re almost certainly wrong. With all of the other features covered in this chapter, there’s little need to ever use this cmdlet. Certainly, you should never call it on unvalidated user input. Incorrect use of this cmdlet can and has led to code-injection attacks and other security issues in the wild. You have been warned. We will now return you to your regularly scheduled section.

Here’s an example where you build a string with several statements in it and execute it:

PS> $expr = '$a=10;'

PS> $expr += 'while ($a--) { $a }'

PS> $expr += '"A is now $a"'

PS> [string](Invoke-Expression $expr)

9 8 7 6 5 4 3 2 1 0 A is now -1

The first three commands in this example build a string to execute. The first line initializes the variable $a, the second adds a while loop that decrements and outputs $a, and the third line outputs a string telling you the final value of $a. Note the double quoting in the last script fragment. Without the nested double quotes, it would try to execute the first word in the string instead of emitting the whole string.

### 10.8.2 The ExecutionContext Variable

One of the predefined variables (also called automatic variables) provided by the PowerShell engine is $ExecutionContext. This variable is another way to get at various facilities provided by the PowerShell engine. It’s intended to mimic the interfaces available to the cmdlet author. The services that matter most to us in this chapter are those provided through the InvokeCommand member. Let’s look at what this member can do for us:

PS> $ExecutionContext.InvokeCommand | Get-Member

TypeName: System.Management.Automation.

CommandInvocationIntrinsics

Name MemberType Definition

---- ---------- ----------

Equals Method bool Equals(System...

ExpandString Method string ExpandStrin...

GetCmdlet Method System.Management...

GetCmdletByTypeName Method System.Management...

GetCmdlets Method System.Collections...

GetCommand Method System.Management...

GetCommandName Method System.Collections...

GetCommands Method System.Collections...

GetHashCode Method int GetHashCode()

GetType Method type GetType()

InvokeScript Method System.Collections...

NewScriptBlock Method scriptblock NewScr...

ToString Method string ToString()

CommandNotFoundAction Property System.EventHandle...

HasErrors Property bool HasErrors {ge...

PostCommandLookupAction Property System.EventHandle...

PreCommandLookupAction Property System.EventHandle...

The interesting methods in this list are ExpandString(), InvokeScript(), and NewScriptBlock(). These methods are covered in the next few sections.

### 10.8.3 The ExpandString() Method

The ExpandString() method lets you perform the same kind of variable interpolation that the PowerShell runtime does in scripts. Here’s an example. First, set $a to a known quantity:

PS> $a = 13

Next, create a variable $str that will display the value of $a:

PS> $str='a is $a'

Because the variable was assigned using single quotes, no string expansion took place. You can verify this by displaying the string:

PS> $str

a is $a

Now call the ExpandString() method, passing in $str:

PS> $ExecutionContext.InvokeCommand.ExpandString($str)

a is 13

It returns the string with the variable expanded into its value. An obvious use for this API is templating. If you have a text file containing PowerShell variables and subexpressions, you can cause those variables to be expanded on the file by doing

PS> $ExecutionContext.InvokeCommand.ExpandString((Get-Content templatefile.txt))

The only downside to this technique is that you need to know what variables are in the file so you can make sure they’re all set properly before expanding the template.

### 10.8.4 The InvokeScript() Method

InvokeScript() does the same thing that the Invoke-Expression cmdlet does (in fact, the cmdlet calls the method). It takes its argument and evaluates it like a script. Call this method passing in the string “2+2”

PS> $ExecutionContext.InvokeCommand.InvokeScript('2+2')

4

and it will return 4.

### 10.8.5 Mechanisms for Creating Scriptblocks

The NewScriptBlock() method, like InvokeScript(), takes a string, but instead of executing it, it returns a scriptblock object that represents the compiled script. Let’s use this method to turn the string '1..4 | foreach {$\_ \* 2}' into a scriptblock:

PS> $sb = $ExecutionContext.InvokeCommand.NewScriptBlock(

'1..4 | foreach {$\_ \* 2}')

You save this scriptblock into a variable, so let’s look at it. Because the ToString() on a scriptblock is the code of the scriptblock, you see the code that makes up the body of the scriptblock:

PS> $sb

1..4 | foreach {$\_ \* 2}

Now execute the scriptblock using the & call operator:

PS> & $sb

2

4

6

8

The scriptblock executes, printing out the even numbers from 2 to 8.

There’s a simpler way of doing this by using a static method on the ScriptBlock class. Here’s how to use this static factory class:

PS> $sb = [scriptblock]::Create('1..4 | foreach {$\_ \* 2}')

PS> & $sb

2

4

6

8

Using the [ScriptBlock] type accelerator, the newer mechanism is significantly simpler than the rather long expression in the earlier example.

#### Note

Many people have asked why the PowerShell team doesn’t allow you to cast a string to a scriptblock. The reason is that they want to make the system resilient against code-injection attacks by minimizing the number of places where executable code can be injected into the system. They particularly want code creation to be an explicit act. Casts are more easily hidden, leading to accidental code injections, particularly when the system may prompt for a string. You don’t want those user-provided strings to be converted into code without some kind of check. See the warning at the beginning of this section for more details.

### 10.8.6 Creating Functions Using the Function: Drive

The final way to create a scriptblock is a side effect of creating elements in the function: drive. Earlier you saw that it’s possible to create a named function by assigning a scriptblock to a name in the function: drive:

PS> $function:foo = {'Hello there'}

PS> foo

Hello there

You could also use the Set-Item or New-Item cmdlet to do this. For example:

PS> New-Item function:foo -Value {'Hi!'}

New-Item : The item at path 'foo' already exists.

At line:1 char:1

+ New-Item function:foo -value {'Hi!'}

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidArgument: (:)

[New-Item], PSArgumentException

+ FullyQualifiedErrorId : Argument,Microsoft.PowerShell.Commands.NewItemCommand

You receive an error because the function already exists, so use the -Force parameter to overwrite the existing definition:

PS> New-Item function:foo -Value { 'Hi!' } -Force

CommandType Name Version Source

----------- ---- ------- ------

Function foo

New-Item returns the item created, so you can see that the function has been changed. But that’s using scriptblocks. What happens if you pass in strings? The interpreter will compile these strings into scriptblocks and then assign the scriptblocks to the name. Here’s an example where the body of the function is determined by the expanded string:

PS> $x=5

PS> $y=6

PS> $function:foo = "$x\*$y"

PS> foo

30

PS> $function:foo

5\*6

The variables $x and $y expand into the numbers 5 and 6 in the string, so the resulting scriptblock is

{5\*6}

Now define another function using foo, but add more text to the function:

PS> New-Item function:bar -Value "$function:foo\*3"

CommandType Name Version Source

----------- ---- ------- ------

Function bar

PS> bar

90

In the expanded string, $function:foo expands into “5\*6”, so the new function bar is assigned a scriptblock {5\*6\*3}.

This finishes our discussion of the techniques PowerShell provides for compiling script code at runtime. In the next section, we’ll look at how to embed static languages like C# and Visual Basic in your scripts. This ability to embed fragments of C# or Visual Basic vastly increases what can be done directly with scripts, but at the cost of some increase in complexity.

## 10.9 Compiling Code with Add-Type

In the previous section, we covered techniques for compiling script code at runtime. In this section, you’ll learn how to inline code written in static languages into your scripts. The key to doing this is the Add-Type cmdlet. With the Add-Type cmdlet, you can embed code fragments written in compiled languages like C# or Visual Basic in your scripts and then compile that code when the scripts are loaded.

#### Note

The need for the Add-Type cmdlet is significantly reduced now that you can create your own classes right in PowerShell, but there are some situations, like interoperating with the operating system, where you still need to use a compiled language like C#.

A particularly interesting application of this technique is that you can create dynamic binary modules. This combines some of the best aspects of script modules with binary modules.

Add-Type can also be used to dynamically load existing .NET assemblies at runtime. Finally, this cmdlet can be used to simplify writing scripts that compile static language code into libraries or executables.

### 10.9.1 Defining a New .NET Class: C#

Let’s jump into an example where you’ll dynamically add a new .NET class at runtime using C#. You’ll write the code for this class using C#.

#### Note

Creating a class can also be done directly in PowerShell v5, as you’ll see in [chapter 19](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-19-143.xhtml#ch19). The example in [listing 10.8](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-109-91.xhtml#ch10ex08) provides a technique for older versions of PowerShell.

It’s a simple class, so even if you aren’t a C# programmer, you should be able to understand the code.

#### Listing 10.8: Creating a class using C#

Add-Type @'

using System;

public static class Example1

{

public static string Reverse(string s)

{

Char[] sc = s.ToCharArray();

Array.Reverse(sc);

return new string(sc);

}

}

'@

This command should run with no errors. Once it’s run, use the new type that you’ve added:

PS> [example1]::Reverse('hello there')

ereht olleh

And there you go. You now have a new method for reversing strings. You could also have saved the file externally and then loaded it at runtime.

### 10.9.2 Defining a New Enum at Runtime

An enum type in .NET is a way of creating a fixed set of name-value constants. The ability to define these types is missing from PowerShell prior to version 5.0 (see [chapter 19](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-19-143.xhtml#ch19)), but you can work around this by using Add-Type. You’ll define an enum that can be used to specify a coffee order. You’ll constrain the types of coffee orders you’ll allow to Latte, Mocha, Americano, Cappuccino, or Espresso. First, set a variable to the list of drink types:

PS> $beverages = 'Latte, Mocha, Americano, Cappuccino, Espresso'

Pass a string to Add-Type that contains the fragment of C# needed to define an enum type:

PS> Add-Type "public enum BeverageType { $beverages }"

It should be easy to see what’s going on. You’re defining a public type called BeverageType using the list of drinks in $beverages. Now that you have the type defined, you can use it in a function to create new drink orders:

PS> function New-DrinkOrder ([BeverageType] $beverage)

{

"A $beverage was ordered"

}

This function uses the enum to constrain the type of the argument to the function and then return a string showing what was ordered. Use the function to order a latte:

PS> New-DrinkOrder latte

A Latte was ordered

And the order goes through. Notice that the casing of the drink name matches what was in the DrinkOrder enum definition, not what was in the argument. This is because the argument contains an instance of the DrinkOrder type and not the original string. Let’s try to order something other than a coffee and see what happens:

PS> New-DrinkOrder coke

New-DrinkOrder : Cannot process argument transformation on parameter

'beverage'. Cannot convert value "coke" to type "BeverageType".

Error: "Unable to match the identifier name coke to a valid enumerator name.

Specify one of the following enumerator names and try again:

Latte, Mocha, Americano, Cappuccino, Espresso"

At line:1 char:16

+ New-DrinkOrder coke

+ ~~~~

+ CategoryInfo : InvalidData: (:) [New-DrinkOrder],

ParameterBindingArgumentTransformationException

+ FullyQualifiedErrorId : ParameterArgumentTransformationError,

New-DrinkOrder

This results in a somewhat verbose but helpful error message telling you why the order failed and what the valid drink types are. That’s all well and good, but the customer wants a Coke. Modify the enum definition to include Coke in the list of beverages:

PS> $beverages += ", Coke"

And call Add-Type again:

PS> Add-Type "public enum BeverageType { $beverages }"

Add-Type : Cannot add type. The type name 'BeverageType' already exists.

At line:1 char:1

+ Add-Type "public enum BeverageType { $beverages }"

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidOperation: (BeverageType:String)

[Add-Type], Exception + FullyQualifiedErrorId : TYPE\_ALREADY\_

EXISTS,Microsoft.PowerShell.Commands.AddTypeCommand

This time it fails. Remember what we said about static types: once they’re defined, they can’t be changed. This is something to consider when using Add-Type to inline static code in a script. Static type definitions mean that the script isn’t as easy to update as a normal PowerShell-only script. Now let’s look at how Add-Type can be combined with dynamic modules.

### 10.9.3 Dynamic Binary Modules

Like dynamic script modules, a dynamic binary module is constructed in memory rather than loaded from disk. This is where Add-Type comes in. The content of a binary module is defined by a compiled assembly, not script text, and Add-Type lets you build these in-memory assemblies. This script constructs a binary module.

#### Listing 10.9: The ExampleModuleScript

$code = @' 1

using System.Management.Automation;

[Cmdlet("Write", "InputObject")]

public class MyWriteInputObjectCmdlet : Cmdlet

{

[Parameter()]

public string Parameter1;

[Parameter(Mandatory = true, ValueFromPipeline=true)]

public string InputObject;

protected override void ProcessRecord()

{

if (Parameter1 != null)

WriteObject(Parameter1 + ":" + InputObject);

else

WriteObject(InputObject);

}

}

'@

$bin = Add-Type $code -PassThru 2

$bin.Assembly | Import-Module 3

* 1 Contains cmdlet code
* 2 Compiles code in memory
* 3 Gets assembly ref from type

This script packages the C# code for a cmdlet into a here-string 1. It then uses Add-Type to produce the required in-memory assembly 2, which it passes to Import-Module 3. The one wrinkle in this approach is the fact that Add-Type returns type objects, not assemblies. Fortunately, this is easy to work around: The type object makes its containing assembly available through the Assembly property. Let’s try out the script. First, load it:

PS>./ExampleModuleScript

then check to see if the module has been created:

PS> Get-Module

ModuleType Version Name ExportedCommands

---------- ------- ---- ----------------

Binary 0.0.0.0 dynamic\_code\_module... Write-InputObject

and there it is. Next, get the CommandInfo object for the new cmdlet:

PS> $cmd = Get-Command Write-InputObject

PS> $cmd | Format-List

Name : Write-InputObject

CommandType : Cmdlet

Definition :

Write-InputObject -InputObject <string>

[-Parameter1 <string>] [<CommonParameters>]

Path :

AssemblyInfo :

DLL :

HelpFile : -Help.xml

ParameterSets : {-InputObject <string> [-Parameter1 <string>]

[<CommonParameters>]}

ImplementingType : MyWriteInputObjectCmdlet

Verb : Write

Noun : InputObject

Notice that the Path, DLL, and AssemblyInfo fields for this command are empty. Because the assembly for a dynamic binary module is in-memory only, these items are empty. They need an assembly that was loaded from disk in order to be defined.

Dynamic binary modules make it possible to get the advantages of a script module (being able to read the script) along with the advantages of compiled code (speed and static type checking). The only disadvantage to the user compared with regular binary modules is that the load time may be a bit longer.

## 10.10 Summary

* Metaprogramming is a set of powerful techniques that cracks open the PowerShell runtime.
* Metaprogramming allows you to extend the runtime with new keywords and control structures.
* You can directly add properties and methods to objects in PowerShell to adapt or extend objects logically in specific problem domains.
* The fundamental unit of PowerShell code, including the content of all functions, scripts, and modules, is the scriptblock.
* Scriptblocks let you define methods that can be added to objects as script methods.
* Scriptblocks don’t necessarily need to be named, and they can be used in many situations, including as the content of variables.
* Scriptblocks are the key to all of the metaprogramming features in PowerShell; they’re also an everyday feature that users work with all the time when they use the ForEach-Object and Where-Object cmdlets.
* The call operator & allows you to invoke commands indirectly, by reference rather than by name (a scriptblock is a reference). This also works with the CommandInfo objects returned from Get-Command.
* When using the Update-TypeData cmdlet, you can load type configuration files that allow you to extend a type instead of a single instance of that type.
* PowerShell supports the use of little language, or domain-specific language techniques, to extend the core language. This allows you to more naturally specify solutions for problems in a particular domain.
* You can employ a variety of techniques for compiling and executing code at runtime. You can use the Invoke-Expression cmdlet, engine invocation intrinsics on the $ExecutionContext variable, or the CreateScriptBlock() static method on the [scriptblock] type.
* Dynamic modules allow you to do local isolation in a script. They also underlie the implementation of closures in PowerShell and provide a simpler way to create custom objects.
* The Add-Type cmdlet lets you work with compiled languages from within PowerShell.
* Add-Type provides a means to embed code in these languages directly in your scripts. This ability adds significant power to the environment at some cost in complexity.
* Add-Type also makes it possible to create dynamic binary modules, allowing you to combine some of the benefits of both static and dynamic coding techniques.

This finishes our look at modules and metaprogramming. In the next chapter, we’ll move on to examine the techniques you can use to work with remote machines.

## Chapter 11: PowerShell Remoting

### Overview

This chapter covers

* Commands with built-in remoting
* PowerShell remoting subsystem
* Using PowerShell remoting
* Remoting sessions, persistent connections, and implicit remoting
* Remoting considerations and custom remoting sessions

*In a day when you don’t come across any problems, you can be sure that you are traveling in the wrong path.  
Swami Vivekananda*

PowerShell is a tool intended for enterprise and cloud management but if it can’t manage distributed systems it isn’t useful. Fortunately, PowerShell has a comprehensive built-in remoting subsystem. This facility allows you to handle most remoting tasks in any kind of configuration you might encounter.

In this chapter, we’re going to cover the features of remoting and how you can apply them. We’ll use an example showing how to combine the features to solve a nontrivial problem: monitoring multiple remote machines. We’ll then look at some of the configuration considerations you need to be aware of when using PowerShell remoting.

Let’s start with a quick overview of PowerShell remoting.

## 11.1 PowerShell Remoting Overview

The ultimate goal for remoting is to be able to execute a command on a remote computer. There are two ways to approach this. First, you could have each command do its own remoting. In this scenario, the command is still executed locally but uses system-level networking capabilities like DCOM to perform remote operations. A number of commands do this, which we’ll cover in the next section. The negative aspect of this approach is that each command has to implement and manage its own remoting and authentication mechanisms.

PowerShell includes a second, more general solution, allowing you to send a command (or pipeline of commands or even a script) to the target machine for execution and then retrieve the results. With this approach, you only have to implement the remoting mechanism once and then it can be used with any command. This second solution is the one we’ll spend most of our time discussing. But first let’s look at the commands that implement their own remoting.

### 11.1.1 Commands with Built-in Remoting

A number of commands in PowerShell have a -ComputerName parameter, which allows you to specify the target machine to access. You can discover (some of) these cmdlets by running either of these commands:

PS> Get-Help \* -Parameter ComputerName

PS> Get-Command -ParameterName ComputerName

#### CIM sessions

Common Information Model (CIM) sessions (see [chapter 16](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-16-125.xhtml#ch16)) are closely related to PowerShell remoting—they enable more efficient access to WMI classes on remote machines. The cmdlets capable of using CIM sessions can be discovered in a similar way:

PS>Get-Command -ParameterName Cimsession

For a new PowerShell v5.1 session on Windows 10, the majority of the cmdlets are listed in [table 11.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-111-94.xhtml#ch11table01).

#### Note

The number of cmdlets you see will depend on the modules you have on your machine. You won’t necessarily see identical results from the two commands given earlier because Get-Help is dependent on analyzing the help files.

#### Note

We’ve deliberately excluded the \*WSMan\* cmdlets from [table 11.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-111-94.xhtml#ch11table01). The \*WSMan\* cmdlets are effectively deprecated and have been replaced by the \*-CIM\* cmdlets (see [chapter 16](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-16-125.xhtml#ch16)).

|  |  |  |
| --- | --- | --- |
| **Table 11.1: Cmdlets with built-in remoting capability** | | |
| Add-Computer | Clear-EventLog | Connect-PSSession |
| Enter-PSSession | Get-EventLog | Get-HotFix |
| Get-Process | Get-Service | Get-WmiObject |
| Invoke-Command | Invoke-WmiMethod | Limit-EventLog |
| New-EventLog | New-PSSession | Receive-Job |
| Receive-PSSession | Register-WmiEvent | Remove-Computer |
| Remove-EventLog | Remove-PSSession | Remove-WmiObject |
| Rename-Computer | Restart-Computer | Send-MailMessage |
| Set-DscLocalConfigurationManager | Set-Service | Set-WmiInstance |
| Show-EventLog | Start-DscConfiguration | Stop-Computer |
| Test-Connection | Write-EventLog |  |

These commands do their own remoting because either the underlying infrastructure already supports remoting or they address scenarios that are of particular importance to system management. You need to supply only one or more computer names to use them against a remote target:

PS> Get-Service -Name BITS -ComputerName W16TGT01, W16DSC02

Status Name DisplayName

------ ---- -----------

Stopped BITS Background Intelligent Transfer Service

Stopped BITS Background Intelligent Transfer Service

You don’t get any indication of which result belongs to which machine by default. In this case, you need to include the MachineName property in the output:

PS> Get-Service -Name BITS -ComputerName W16TGT01, W16DSC02 |

select Status, Name, MachineName

Status Name MachineName

------ ---- -----------

Stopped BITS W16DSC02

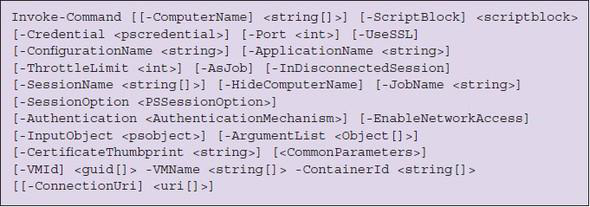
Stopped BITS W16TGT01

Self-remoting is performed using DCOM and RPC. These protocols will be blocked by default by firewalls. Also, the set of commands that do self-remoting is quite small, so the remaining commands must rely on the PowerShell remoting subsystem to access remote computers. We’ll start looking at that in the next section.

### 11.1.2 The PowerShell Remoting Subsystem

You’ve seen a few brief examples of how remoting works in previous chapters. You may remember that all those examples used the same basic cmdlet: Invoke-Command. This cmdlet allows you to remotely invoke a scriptblock on another computer and is the building block for most of the features in remoting. The partial syntax for this command is shown in [figure 11.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-111-94.xhtml#ch11fig01).

Figure 11.1: Partial syntax for the Invoke-Command cmdlet, which is the core of PowerShell’s remoting capabilities. This cmdlet is used to execute commands and scripts on one or more computers. It can be used synchronously or asynchronously as a job. The VMId, VMName, and ContainerId parameters were introduced with PowerShell 5.1 and are valid only on Windows 10 and Windows Server 2016 (or later)



The Invoke-Command cmdlet is used to invoke a scriptblock on one or more computers. You do so by specifying a computer name (or list of names) for the machines on which you want to execute the command. For each name in the list, the remoting subsystem will take care of all the details needed to open the connection to that computer, execute the command, retrieve the results, and then shut down the connection. If you’re going to run the command on a large set of computers, Invoke-Command will also take care of all resource management details, such as limiting the number of concurrent remote connections. Our previous example becomes this:

PS> Invoke-Command -ScriptBlock {Get-Service -Name BITS} `

-ComputerName W16TGT01, W16DSC02

Status Name DisplayName PSComputerName

------ ---- ----------- --------------

Stopped BITS Background Intelligent Transfer Service W16DSC02

Stopped BITS Background Intelligent Transfer Service W16TGT01

Note that you now get the computer name that the result refers to in the output.

This is a simple but powerful model if you need to execute only a single command or script on the target machine. But if you want to execute a series of commands on the target, the overhead of setting up and taking down a connection for each command becomes expensive. PowerShell remoting addresses this situation by allowing you to create a persistent connection to the remote computer called a session. You do so by using the New-PSSession cmdlet.

Both of the scenarios we’ve discussed so far involve what is called noninteractive remoting because you’re only sending commands to the remote machines and then waiting for the results. You don’t interact with the remote commands while they’re executing.

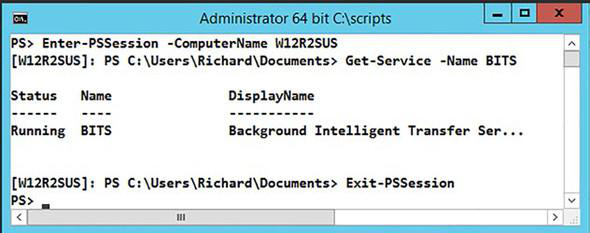
Another standard pattern in remoting occurs when you want to set up an interactive session where every command you type is sent transparently to the remote computer. This is the style of remoting implemented by tools like Remote Desktop, Telnet, or SSH (Secure Shell).

#### Note

The PowerShell team has announced that SSH support will be built into PowerShell. Basic terminal support will be available with Windows Server 2016. Full SSH integration with the PowerShell Remoting Protocol will be introduced at a later date. [Appendix A](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/appendix-158.xhtml#app01) demonstrates SSH-based remoting between Linux and Windows machines using PowerShell v6.

PowerShell allows you to start an interactive session using the Enter-PSSession cmdlet. Use Exit-PSSession to close the session when you've finished working. If you enter a remote session created by New-PSSession, then using Exit-PSSession will suspend the session without closing the remote connection. Because the connection isn’t closed, you can later reenter the session with all session data preserved by using Enter-PSSession again. An example of an interactive session is given in [figure 11.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-111-94.xhtml#ch11fig02).

Figure 11.2: Interactive remoting session to the computer W12R2SUS. Notice how the PowerShell prompt changes to incorporate the remote machine name when you enter the session



These cmdlets—Invoke-Command, New-PSSession, and Enter-PSSession—are the basic remoting tools you’ll be using. But before you can use them, you need to make sure remoting is enabled, so we’ll look at that next.

### 11.1.3 Enabling Remoting

At this point we have some good news and some bad news for you. The good news is that for Windows Server 2012 and later (including Windows Server 2012 R2 virtual machines running in Azure IaaS), PowerShell remoting is enabled by default. The bad news is that for earlier versions of Window Server and for all versions of the Windows client operating system, PowerShell remoting is turned off by default and has to be enabled.

#### Note

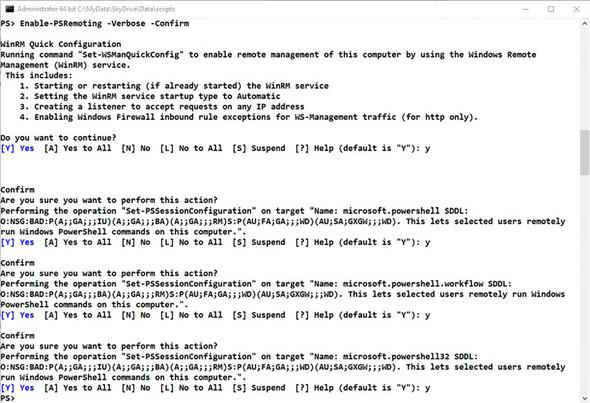
You have to turn on PowerShell remoting for a machine to receive and execute remote administration commands. You don’t need to turn on remoting to send commands, though you will need to turn it on at least temporarily to change client-side settings such as the TrustedHosts list on the local machine.

You enable remoting using the Enable-PSRemoting cmdlet. To run this command, you must have administrator privileges on the machine you’re going to enable. You need to do the following:

* Start the PowerShell session with elevated privileges (Run As Administrator).
* Ensure that none of the network connections on the machine has a network profile of Public. Use Get-NetConnectionProfile | Set-NetConnectionProfile -NetworkCategory Private to set the network profile.

By default, Enable-PSRemoting runs silently with no output and no input required. You can use the –Verbose and –Confirm parameters to see what’s happening, as shown in [figure 11.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-111-94.xhtml#ch11fig03).

Figure 11.3: Enabling PowerShell remoting on a machine



The Enable-PSRemoting command performs all the configuration steps needed to allow users with local administrator privileges to remote to this computer in a domain environment. In a non-domain or workgroup environment, as well as for non-admin users, additional steps are required for remoting to work.

### 11.1.4 Additional Setup Steps for Workgroup Environments

If you’re working in a workgroup environment—for example, at home—you must take a few additional steps before you can connect to a remote machine. With no domain controller available to handle the various aspects of security and identity, you have to manually configure the names of the computers you trust. If you want to connect to the computer computerItrust, then you have to add it to the list of trusted computers (or TrustedHosts list).

You can do this via the WSMan: drive, as shown in [table 11.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-111-94.xhtml#ch11table02). Note that you need to be running as administrator to be able to use the WSMan: provider. Once you’ve completed these steps, you’re ready to start playing with some examples.

#### A note on security

The computers in the TrustedHosts list are implicitly trusted by the local computer when you add their names to this list. It’s not an incoming security feature like a firewall. The identity of these computers won’t be authenticated when you connect to them. Because the connection process requires sending credential information to these machines, you need to be sure that you can trust these computers. Also, be aware that the TrustedHosts list on a machine applies to everyone who uses that computer, not only the user who changed the setting.

That said, unless you allow random people to install computers on your internal network, this shouldn’t introduce substantial risk most of the time. If you’re comfortable with knowing which machines you’ll be connecting to, you can put \* in the TrustedHosts list, indicating that you’re implicitly trusting any computer you might be connecting to. As always, security is a principle tempered with pragmatics.

| **Table 11.2: Additional steps needed to enable remote access to a computer in a workgroup environment** | | |
| --- | --- | --- |
| **Step** | **Command** | **Description** |
| 1 | cd wsman:\localhost\client | cd’ing into the client configuration node in the WSMan: drive allows you to access the WS-MAN configuration for this computer using the provider cmdlets. |
| 2 | $old = (Get-Item .\TrustedHosts).Value | You’ll want to update the current value of the TrustedHosts item, so you get it and save the value in a variable. |
| 3 | $old += ‘,computerItrust’ | The value of TrustedHosts is a string containing a comma-separated list of the computers considered trustworthy. You add the new computer name to the end of this list, prefixed with a comma. (If you’re comfortable with implicitly trusting any host, then set this string to \*, which matches any hostname.) |
| 4 | Set-Item .\TrustedHosts $old | Once you’ve verified that the updated contents of the variable are correct, you assign it back to the TrustedHosts item, which updates the configuration. |

An alternative way of validating the identity of the target computer is to use HTTPS when connecting to that computer. This works because, in order to establish an HTTPS connection, the target server must have a valid certificate installed where the name in the certificate matches the server name. As long as the certificate is signed by a trusted certificate authority you know that the server is the one it claims to be. Unfortunately, this process does require that you have a valid certificate, issued by either a commercial or local CA. This is an entirely reasonable requirement in an enterprise environment but may not always be practical in smaller or informal environments.

### 11.1.5 Authenticating the Connecting User

In the previous section, you saw how the client verifies the identity of the target computer. Now we’ll explore the converse of this—how the target computer verifies the identity of the connecting user. PowerShell remoting supports a wide variety of ways of authenticating a user, including NTLM and Kerberos. Each mechanism has its advantages and disadvantages. The authentication mechanism also has an important impact on how data is transmitted between the client and the server. Depending on how you authenticate to the server, the data passed between the client and server may or may not be encrypted. Encryption is extremely important in that it protects the contents of your communications with the server against tampering and preserves privacy. If encryption isn’t being used, you need to ensure the physical security of your network. No untrusted access to the network can be permitted in this scenario. The possible types of authentication are shown in [table 11.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-111-94.xhtml#ch11table03).

| **Table 11.3: Possible types of authentication available for PowerShell remoting** | | |
| --- | --- | --- |
| **Auth type** | **Description** | **Encrypted payload** |
| Default | Use the authentication method specified by the WS-Management Protocol. | Depends on what was specified. |
| Basic | Use Basic Authentication, part of HTTP, where the username and password are sent unencrypted to the target server or proxy. | No. Use HTTPS to encrypt the connection. |
| Digest | Use Digest Authentication, which is also part of HTTP. This mechanism supersedes Basic Authentication and encrypts the credentials. | Yes. |
| Kerberos | The client computer and the server mutually authenticate using the Kerberos network authentication protocol. | Yes. |
| Negotiate | Negotiate is a challenge-response scheme that negotiates with the server or proxy to determine the scheme to use for authentication. For example, negotiation is used to determine whether the Kerberos protocol or NTLM is used. | Yes. |
| CredSSP | Use Credential Security Service Provider (CredSSP) authentication, which allows the user to delegate credentials. This mechanism, introduced with Windows Vista, is designed to support the second-hop scenario, where commands that run on one remote computer need to hop to another computer to do something. | Yes. |

For all the authentication types except Basic, the payload of the messages you send is encrypted directly by the remoting protocol. If Basic authentication is chosen, you have to use encryption at a lower layer—for example, by using HTTPS instead of HTTP.

### 11.1.6 Enabling Remoting in the Enterprise

Remote administration is most likely to be performed against the servers in your environment. As you’ve seen, the newer versions of Windows Server have PowerShell remoting enabled by default. If you have older servers, you don’t want to have to enable remoting on them individually because you may be dealing with tens, hundreds, or thousands of machines. Obviously, you can’t use PowerShell remoting to turn on remoting, so you need another way to push configuration out to a collection of machines. This is exactly what Group Policy is designed for. You can use Group Policy to enable and configure remoting as part of the machine policy that gets pushed out.

PowerShell depends on the WinRM (Windows Remote Management) service for its operation. Your Group Policy needs to:

* Ensure the WinRM service will start automatically and is started.
* Configure WinRM to accept remoting requests.
* Configure Windows Firewall to allow remoting requests.

Instructions on creating a suitable Group Policy are available at <http://mng.bz/3aHW>.

## 11.2 Applying PowerShell Remoting

With remoting services enabled, you can start to use them to get your work done. In this section, we’re going to look at ways you can apply remoting to solve management problems. We’ll start with some simple remoting examples. Next, we’ll work with more complex examples where we introduce concurrent operations. Then you’ll apply the principles you’ve learned to solve a specific problem: how to implement a multi-machine configuration monitor. You’ll work through this problem in a series of steps, adding more capabilities to your solution, resulting in a simple but fairly complete configuration monitor. Let’s start with the most basic examples.

### 11.2.1 Basic Remoting Examples

Building on our “Hello world” example from [chapter 1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-1-8.xhtml#ch01), the most basic example of remoting is

Invoke-Command -ComputerName Servername -ScriptBlock {'Hello world'}

The first thing to notice is that Invoke-Command takes a scriptblock to specify the actions. This pattern should be familiar by now—you’ve seen it with ForEach-Object and Where-Object many times. The Invoke-Command does operate a bit differently, though. It’s designed to make remote execution as transparent as possible. For example, if you want to sort objects, the local command looks like this:

PS> 1..3 | sort -Descending

Now if you want to do the sorting on the remote machine, you’d do this:

PS> 1..3 |

Invoke-Command -ComputerName localhost -ScriptBlock {sort -Descending}

You’re splitting the pipeline across local and remote parts, and the scriptblock is used to demarcate which part of the pipeline should be executed remotely.

#### Note

Localhost is used to set a remote session to your local machine for testing purposes. You could use the machine name if preferred or $ENV:COMPUTERNAME.

This works the other way as well:

PS> Invoke-Command -ComputerName localhost -ScriptBlock { 1..3 } |

sort -Descending

Here you’re generating the numbers on the remote computer and sorting them locally. Scriptblocks can contain more than one statement. This implies that the semantics need to change a bit. Whereas in the simple pipeline case streaming input into the remote command was transparent, when the remote command contains more than one statement, you have to be explicit and use the $input variable to indicate where you want the input to go. That looks like the following:

PS> 1..3 | Invoke-Command -ComputerName localhost -ScriptBlock {

'First'

$input | sort -Descending

'Last'

}

First

3

2

1

Last

The scriptblock argument to Invoke-Command in this case contains three statements. The first emits the string 'First', the second does the sort on the input, and the third emits the string 'Last'.

What happens if you don’t specify input? Nothing is emitted between 'First' and 'Last'. Because $input wasn’t specified, the input objects were never processed. You’ll need to keep this in mind when you start to build a monitoring solution.

Now let’s look at how concurrency—multiple operations occurring at the same time—impacts your scripts.

### 11.2.2 Adding Concurrency to the Examples

In [chapter 1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-1-8.xhtml#ch01), we talked about how each object passed completely through all states of a pipeline, one by one. This behavior changes with remoting because the local and remote commands run in separate processes that are executing concurrently. This means you now have two threads of execution—local and remote—and that can have an effect on the order in which things are executed. Consider the following statement:

PS> 1..3 | foreach { Write-Host $\_ -ForegroundColor green;

$\_; Start-Sleep 5 } | Write-Host

1

1

2

2

3

3

This statement sends a series of numbers down the pipeline. In the body of the foreach scriptblock, the value of the current pipeline object is written to the screen (in green) and then passed to the next state in the pipeline. This last stage also writes the object to the screen (in standard color). Given that you know each object is processed completely by all stages of the pipeline, the order of the output is as expected. The first number is passed to foreach, where it’s displayed and then passed to Write-Output, where it’s displayed again, so you see the sequence 1, 1, 2, 2, 3, 3.

#### Note

Start-Sleep is used to build sufficient pauses into the execution so that you can see what’s happening. Run the code without Start-Sleep to see the difference.

Now let’s run this command again using Invoke-Command in the final stage:

PS> 1..3 | foreach {

Write-Host -ForegroundColor green $\_

$\_; Start-Sleep 5 } |

Invoke-Command -ComputerName localhost -ScriptBlock { Write-Host }

1

2

1

3

2

3

The order has changed—you see 1 and 2 from the local process in green on a color display, then you see 1 from the remote process (in your normal foreground text color), and so on. The local and remote pipelines are executing at the same time, which is what’s causing the changes to the ordering. Predicting the order of the output is made more complicated by the use of buffering and timeouts in the remoting protocol.

You used the Start-Sleep command in these examples to force these visible differences. If you take out this command, you’ll get a different pattern:

PS> 1..3 | foreach { Write-Host $\_ -ForegroundColor green ; $\_ } |

Invoke-Command -ComputerName localhost -ScriptBlock { Write-Host }

1

2

3

1

2

3

This time, all the local objects are displayed (in green) and then passed to the remoting layer, where they’re buffered until they can be delivered to the remote connection. This way, the local side can process all objects before the remote side starts to operate. Concurrent operation and buffering make it appear a bit unpredictable, but if you didn’t have the Write-Hosts in place, it would be unnoticeable. The important thing to understand is that objects being sent to the remote end will be processed concurrently with the local execution. That means the remoting infrastructure doesn’t have to buffer everything sent from the local end before starting execution.

Up to now, you’ve been passing only simple commands to the remote end. But because Invoke-Command takes a scriptblock, you can, in practice, send pretty much any valid PowerShell script. You’ll take advantage of this fact in the next section when you start to build your multi-machine monitor.

#### Note

Why does remoting require scriptblocks? Two reasons: Scriptblocks are always compiled locally so you’ll catch syntax errors as soon as the script is loaded, and using scriptblocks limits vulnerability to code injection attacks by validating the script before sending it.

### 11.2.3 Solving a Real Problem: Multi-Machine Monitoring

In this section, you’re going to build a solution for a real management problem: multi-machine monitoring. With this solution, you’re going to gather some basic health information from the remote host. The goal is to use this information to determine when a server may have problems such as out of memory, out of disk, or reduced performance due to a high faulting rate. You’ll gather the data on the remote host and return it as a hashtable so you can look at it locally.

Your requirements are as follows:

* Collect the amount of free space on the C: drive from the Get-PSDrive command.
* Collect the page fault rate retrieved using CIM (WMI).
* Collect the processes consuming the most CPU from Get-Process with a pipeline.
* Collect the processes that have the largest working set, also from Get-Process.
* Ensure the list of computers you monitor aren’t hardcoded into the script; the computers to monitor will be listed in a file.
* Monitor each computer on specific days with the results stored in the file.
* Apply a throttle limit to control how many simultaneous machines are monitored.
* Parameterize the script for ease of use.

This listing shows a solution to the problem using the techniques you’ve learned so far in the book.

#### Listing 11.1: Parameterized monitoring script

param ( 1

[string] $serverFile = 'servers.txt', 1

[int] $throttleLimit = 10, 1

[int] $numProcesses = 5 1

)

$gatherInformation ={ 2

param ([int] $procLimit = 5)

@{

Date = Get-Date

FreeSpace = (Get-PSDrive c).Free

PageFaults = (Get-WmiObject `

Win32\_PerfRawData\_PerfOS\_Memory).PageFaultsPersec

TopCPU = Get-Process |

Sort-Object CPU -Descending |

Select-Object -First $procLimit

TopWS = Get-Process |

Sort-Object WS -Descending |

Select-Object -First $procLimit

}

}

$servers = Import-CSV $serverfile | 3

Where-Object { $\_.Day -eq (Get-Date).DayOfWeek } |

foreach { $\_.Name }

Invoke-Command -ThrottleLimit $throttleLimit -ComputerName $servers `

-ScriptBlock $gatherInformation `

-ArgumentList $numProcesses 4

* 1 Define parameters
* 2 Create scriptblock
* 3 Get servers to monitor
* 4 Perform monitoring

The first two parameters 1 are obvious: $ServerFile is the name of the file containing the list of servers to check, and $throttleLimit is the throttle limit (number of simultaneous connections the monitoring script makes to remote machines). The default throttle limit for Invoke-Command is 32. We’re deliberately lowering that to ensure we don’t overload the local machine.

The third parameter, $numProcesses, controls the number of process objects to include in the TopCPU and TopWS entries in the table returned from the remote host. Although you could in theory trim the list that gets returned locally, you can’t add to it, so you need to evaluate this parameter on the remote end to get full control. That means it has to be a parameter to the remote command. This is another reason scriptblocks are useful. You can add parameters to the scriptblock that’s executed on the remote end.

The scriptblock to be passed to the remote machines is defined 2. Notice the parameter on the scriptblock that’s executed on the remote end. That’s how the number of processes to return is passed to the remote server.

The list of servers is derived from the input file 3. The contents of servers.txt would look something like this:

Name,Day

W16DSC01,Monday

W16TGT01,Tuesday

W16PWA01,Wednesday

W16DSC02,Saturday

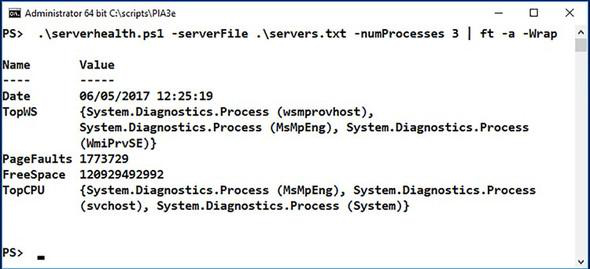
W16CN01,Thursday

W16AS01,Friday

When you load the servers, you’ll do some processing on this list to determine the current day of the week and decide which servers need monitoring.

The final step 4 is to use Invoke-Command to send the scriptblock to the appropriate servers. [Figure 11.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-112-95.xhtml#ch11fig04) shows the script in action.

Figure 11.4: [Listing 11.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-112-95.xhtml#ch11ex01) in action



[Listing 11.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-112-95.xhtml#ch11ex01) was saved as serverhealth.ps1. We decided we needed only the top three processes rather than the default five. The data is returned as a hashtable. Notice that the process data is embedded as objects. You’d need to perform further processing locally if you wanted to drill down into the process objects.

The result is that, with a small amount of code, you’ve created a flexible framework for an agentless distributed health monitoring system. With this system, you can run this health model on any machine without having to worry about whether the script is installed on that machine or whether the machine has the correct version of the script. It’s always available and always the right version because the infrastructure is pushing it out to the target machines. You can even have different files of server names if required.

#### Note

What we’re doing here isn’t what most people would call monitoring, which usually implies a continual semi-real-time mechanism for noticing a problem and then generating an alert. This system is certainly not real time, and it’s a pull model, not a push. This solution is more appropriate for configuration analysis.

You now have an idea of how to use remoting to execute a command on a remote server. This is a powerful mechanism, but sometimes you need to send more than one command to a server; for example, you might want to run multiple data-gathering scripts, one after the other, on the same machine. Because there’s a significant overhead in setting up each remote connection, you don’t want to create a new connection for every script you execute. Instead, you want to be able to establish a persistent connection to a machine, run all the scripts, and then shut down the connection.

## 11.3 PowerShell Remoting Sessions and Persistent Connections

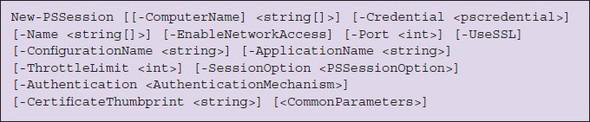
In the previous section, you learned how to run individual scriptblocks on remote machines. From the user’s point of view, the Invoke-Command operation is simple, but under the covers the system has to do a lot of work creating, using, and deleting the connection, which makes creating a new connection each time a costly proposition. Also, you can’t maintain any state—things like variable settings or function definitions—on the remote host.

To address these issues, in this section we’ll show you how to create persistent connections called sessions that will give you much better performance when you want to perform a series of interactions with the remote host as well as allow you to maintain remote state. In the simplest terms, a session is the environment where PowerShell commands are executed. This is true even when you run the console host, PowerShell.exe. The console host program creates a local session that it uses to execute the commands you type. This session remains alive until you exit the program. When you use remoting to connect to another computer, you’re also creating one remote session for every local session you remote from until explicitly closed. An instance of wsmprovhost.exe per connecting session will run on the remote host as long as that session is open.

Each session contains all the things you work with in PowerShell—all the variables, all the functions that are defined, and the history of the commands you typed—and each session is independent of any other session. If you want to work with these sessions, you need a way to manipulate them. You do this in the usual way: through objects and cmdlets. PowerShell represents sessions as objects that are of type PSSession.

By default, every time you connect to a remote computer by name with Invoke -Command, a new PSSession object is created to represent the connection to that remote machine. If you’re going to run more than one command on a computer, you need a way to create persistent connections to that computer. You can do this with New-PSSession; the syntax for this cmdlet is shown in [figure 11.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-113-96.xhtml#ch11fig05).

Figure 11.5: The syntax for the New-PSSession cmdlet. This cmdlet is used to create persistent connections to a remote computer



This command has many of the same parameters that you saw in Invoke-Command. The difference is that, for New-PSSession, these parameters are used to configure the persistent session instead of the transient sessions you saw being created by Invoke -Command. The PSSession object returned from New-PSSession can then be used to specify the destination for the remote command instead of the computer name.

The lifetime of the session begins with the call to New-PSSession and persists until it’s explicitly destroyed by the call to Remove-PSSession. Let’s look at an example that illustrates how much of a performance difference sessions can make. You’ll run Get -Date five times using Invoke-Command and see how long it takes using Measure-Command (which measures command execution time).

First, execute the test without sessions:

PS> Measure-Command { 1..5 |

foreach { Invoke-Command W16TGT01 {Get-Date} } } |

Format-Table -AutoSize TotalSeconds

TotalSeconds

------------

4.7129865

The result from Measure-Command shows that each operation appears to be taking a little under one second. Modify the example to create a session at the beginning and then reuse it in each call to Invoke-Command:

PS> Measure-Command {

$s = New-PSSession W16TGT01

1..5 |

foreach { Invoke-Command $s {Get-Date} }

Remove-PSSession $s

} |

Format-Table -AutoSize TotalSeconds

TotalSeconds

------------

0.8096949

This output shows that it’s taking about one-sixth the time as the first command. Increasing the number of remote invocations from 5 to 50 results in an execution time of 1.4997587 seconds. Clearly, for this simple example, the time to set up and break down the connection totally dominates the execution time. Other factors affect real scenarios, such as network performance, the size of the script, and the amount of information being transmitted. Still, it’s obvious that when multiple interactions are required, using a session will result in substantially better performance.

The downside is that persistent sessions will monopolize your machine’s limited resources, so if you forget to close a session, you may soon hit the limits set (max user connections, max connections per server). Cleaning up unrequired sessions is definitely in your best interest. The two most expensive penalties with remoting are setting up the session and serializing the return data. Filtering on the remote machine to reduce the amount of data to be returned can also significantly improve performance.

### 11.3.1 Additional Session Attributes

This section describes some PSSession attributes that can have an impact on the way you write your scripts.

#### Sessions and Hosts

The host application running your scripts can impact the portability of your scripts if you become dependent on specific features of that host. (This is why PowerShell module manifests include the PowerShellHostName and PowerShellHostVersion elements.) Dependency on specific host functionality is a consideration with remote execution because the remote host implementation is used instead of the normal interactive host. This is necessary to manage the extra characteristics of the remote or job environments. This host shows up as a process named wsmprovhost corresponding to the executable wsmprovhost.exe. This host supports only a subset of the features available in the normal interactive PowerShell hosts.

#### Session Isolation

Another point is the fact that each session is configured independently when it’s created, and once it’s constructed, it has its own copy of the engine properties, execution policy, function definitions, and so on. This independent session environment exists for the duration of the session and isn’t affected by changes made in other sessions. This principle is called isolation—each session is isolated from, and therefore not affected by, any other session.

#### Only One Command Runs at a Time

A final characteristic of a session instance is that you can run only one command (or command pipeline) in a session at one time. If you try to run more than one command at a time, a “session busy” error will be raised. But there’s some limited command queuing: if there’s a request to run a second command synchronously (one at a time), the command will wait up to four minutes for the first command to be completed before generating the “session busy” error. But if a second command is requested to run asynchronously—without waiting—the busy error will be generated immediately.

With some knowledge of the characteristics and limitations of PowerShell sessions, you can start to look at how to use them.

### 11.3.2 Using the New-PSSession Cmdlet

In this section, you’ll learn how to use the New-PSSession cmdlet. Let’s start with an example. First, you’ll create a PSSession on the local machine by specifying localhost as the target computer:

PS> $s = New-PSSession -ComputerName localhost

#### Note

By default a user must be running with elevated privileges to create a session on the local machine. You’ll see how to change the default setting later.

You now have a PSSession object in the $s variable that you can use to execute remote commands. Earlier we said each session runs in its own process. You can confirm this by using the $PID session variable to see what the process ID of the session process is. First, run this code in the remote session

PS> Invoke-Command -Session $s -ScriptBlock {$PID}

9436

and you see that the process ID is 9436. When you get the value in the local session by typing $PID at the command line, as shown here

PS> $PID

8528

you see that the local process ID is 8528.

#### Note

The numbers you see may well be different than those shown here. The important point is that the $PID values are different when running locally and through a remoting session.

Now define a variable in the remote session:

PS> Invoke-Command -Session $s -ScriptBlock {$x=1234}

With this command, you’ve set the variable $x in the remote session to 1234. This works in much the same way as it does in the local case—changes to the remote environment are persisted across the invocations. You can define a function and make it reference the $x variable you defined earlier:

PS> Invoke-Command -Session $s -ScriptBlock {

function hi {"Hello there, x is $x"}

}

PS> Invoke-Command -Session $s -ScriptBlock {hi}

Hello there, x is 1234

You get the preserved value.

#### Note

We’ve had people ask whether other users on the computer can see the sessions we’re creating. As mentioned earlier, this isn’t the case. Users have access only to the remote sessions they create and only from the sessions they were created from. There’s no way for one session to connect to another session that it didn’t itself create. The only aspect of a session that may be visible to another user is the existence of the wsmprovhost process hosting the session.

As you’ve seen, remote execution is like the local case … well, almost. You have to type Invoke-Command every time. If you’re executing a lot of interactive commands on a specific machine, this task quickly becomes annoying. PowerShell provides a much better way to accomplish this type of task, as you’ll see in the next section.

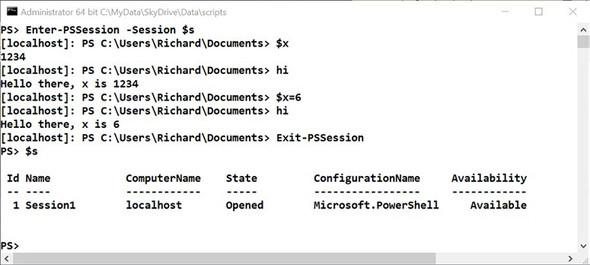
### 11.3.3 Interactive Sessions

In the previous sections, you learned how to issue commands to remote machines using Invoke-Command. This approach is effective but gets annoying for more interactive types of work. To make this scenario easier, you can start an interactive session using the Enter-PSSession cmdlet. Once you’re in an interactive session, the commands you type are automatically passed to the remote computer and executed without having to use Invoke-Command. Let’s try this out. You’ll reuse the session you created in the previous section. In that session, you defined the variable $x and the function hi. To enter interactive mode during this session, you’ll call Enter-PSSession, passing in the session object, as shown in [figure 11.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-113-96.xhtml#ch11fig06).

#### Note

Only interactive commands are transmitted when you use Enter -PSSession. You can’t use it in a script and pass commands to the session.

Figure 11.6: Using a PSSession for interactive remoting



As soon as you enter interactive mode, you see that the prompt changes: it now displays the name of the machine you’re connected to and the current directory.

#### Note

The default prompt can be changed in the remote session in the same way it can be changed in the local session. If you have a prompt definition in your profile, you may be wondering why that wasn’t used. We’ll get to that later when we look at some of the things you need to keep in mind when using remoting.

You can see from the code being run in the figure that the value of $x is preserved (1234) and the hi function you defined is also available. Changing the value of $x and then rerunning the hi function shows the new value displayed in the output.

You can exit an interactive remote session either by using the exit keyword or by using the Exit-PSSession cmdlet. You see that the prompt changed back and the session still exists. It will persist until explicitly removed with Remove-PSSession or the PowerShell instance is closed. You can enter and exit a session as often as you need to as long as it’s not removed in the interim.

Another useful feature to consider is the fact that you can have more than one session open at a time. This means you can pop back and forth between multiple computers as needed, which makes dealing with multiple machines convenient.

More differences exist between the pattern where you used Invoke-Command for each command and the interactive mode. In the non-interactive Invoke-Command case, the remote commands send objects back, where they’re formatted on the local machine. In the interactive remoting case, the objects are formatted on the remote machine, and simple strings are sent to the local machine to be displayed. Usually this won’t matter, but cultural information such as dates and object formatting may be impacted.

Finally, as with the non-interactive remoting case, you can run an interactive session in a temporary session by passing the name of the computer instead of an existing PSSession. Using the PSSession has the advantage that you can enter and exit the remote session and have the remote state preserved between activities. If the name of the computer is passed in, the connection will be torn down when you exit the session. Because a remote session involves creating a remote host process, forgetting to close your sessions can waste resources. At any point, you can use Get-PSSession to get a list of the open sessions you currently have and use Remove-PSSession to close them as appropriate.

By now, you should be comfortable with creating and using persistent remote sessions. What we haven’t spent much time on yet is how to manage all these connections you’re creating.

### 11.3.4 Managing PowerShell Sessions

Each PSSession is associated with an underlying Windows process. As such, it consumes significant resources even when no commands are being executed in it. You should delete PSSessions that are no longer needed. This reduces the memory usage and similar drains on the remote system. At the same time, creating new PSSessions also puts a load on the system, consuming additional CPU resources to create each new process. When managing your resource consumption, you need to balance the cost of creating new sessions against the overhead of maintaining multiple sessions. There’s no hard-and-fast rule for deciding what this balance should be. In the end, you should decide on an application-by-application basis.

To get a list of the existing PSSessions, you use the Get-PSSession command, and to remove sessions that are no longer needed, you use the Remove-PSSession cmdlet. The Remove-PSSession cmdlet closes the PSSession, which causes the remote process to exit and frees up all the resources it held. Removing the session also frees up local resources like the network connection used to connect to the remote session.

With PowerShell v2 you can view the sessions on the local machine, whereas PowerShell v3 and later enable you to see the sessions on remote as well as local machines. On a local machine, you’ll see something like this:

PS> Get-PSSession |

Format-List Id, Name, ComputerName, ComputerType, State,

ConfigurationName, Availability

Id : 1

Name : Session1

ComputerName : W16TGT01

ComputerType : RemoteMachine

State : Opened

ConfigurationName : Microsoft.PowerShell

Availability : Available

The remote machine (use the –ComputerName parameter) may give you results like this:

PS> Get-PSSession -ComputerName W16TGT01 |

Format-List Id, Name, ComputerName, ComputerType, State,

ConfigurationName, Availability

Id : 1

Name : Session1

ComputerName : W16TGT01

ComputerType : RemoteMachine

State : Opened

ConfigurationName : Microsoft.PowerShell

Availability : Available

Id : 3

Name : Session1

ComputerName : W16TGT01

ComputerType : RemoteMachine

State : Disconnected

ConfigurationName : Microsoft.PowerShell

Availability : Busy

In this case, the session with an Id of 1 (state is Opened) is the session created from your local machine. The session with an Id of 3 is another session to the remote machine—in this case, created from a third machine. We know this because we created them. Unfortunately, there’s no way to tell who created a session connected to a remote machine or from which machine it was created. Notice that session Id 3 is shown with a state of Disconnected. This means you aren’t connected to it.

#### Tip

The ID number will change every time you access the sessions on the remote machine created by a PowerShell session other than your own. It’s worth giving your session distinctive names so that you can easily distinguish between sessions.

On the client end, if you don’t explicitly remove the sessions or set timeouts, local sessions will remain open until you end your PowerShell session. But what happens if the client fails for some reason without closing its sessions? If the PowerShell session is closed or the local machine crashes, the remote session will be terminated. If network connectivity is lost or the session times out (the default is two hours), the session may be put into a disconnected state. You can also put a session into a disconnected state manually.

#### Note

Commands continue to run in a disconnected session. You can even deliberately create a disconnected session using the –InDisconnectedSession parameter of Invoke-Command.

The sessions shown earlier in this section have been re-created with distinctive names:

PS> Get-PSSession -ComputerName W16TGT01 |

Format-Table Id, Name, ComputerName, State,

Availability -AutoSize

Id Name ComputerName State Availability

-- ---- ------------ ----- ------------

4 FromW16AS01 W16TGT01 Opened Available

5 FromW16DSC01 W16TGT01 Disconnected Busy

FromW16AS01 is the one from our local machine. That session can be disconnected:

PS> Disconnect-PSSession -Name FromW16AS01

Id Name ComputerName State Availability

-- ---- ------------ ----- ------------

4 FromW16AS01 W16TGT01 Disconnected None

Notice that state changes to Disconnected and availability changes to None. After closing the PowerShell session that created the session FromW16AS01 and opening a new PowerShell session, using Get-PSSession to test for a session will return nothing as expected—we haven’t created any remoting sessions in that PowerShell session.

Now try getting the sessions on the remote server we were working with:

PS> Get-PSSession -ComputerName W16TGT01 |

Format-Table Id, Name, ComputerName, State,

Availability -AutoSize

Id Name ComputerName State Availability

-- ---- ------------ ----- ------------

1 FromW16AS01 W16TGT01 Disconnected None

2 FromW16DSC01 W16TGT01 Disconnected Busy

You can reconnect to the session—in this case session FromW16AS01:

PS> Connect-PSSession -ComputerName W16TGT01

Connect-PSSession : Cannot connect PSSession "FromW16DSC01",

either because it is not in the Disconnected state, or it

is not available for connection.

At line:1 char:1

+ Connect-PSSession -ComputerName W16TGT01

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidOperation: ([PSSession] W16TGT01:PSSession) [Connect-PSSession],

RuntimeExcept ion

+ FullyQualifiedErrorId : PSSessionConnectFailed,Microsoft.PowerShell.Commands.ConnectPSSessionCommand

Id Name ComputerName ComputerType State Availability

-- ---- ------------ ------------ ----- ------------

3 FromW16AS01 W16TGT01 RemoteMachine Opened Available

You can connect to the session FromW16AS01, but you can’t connect to the session from the third machine because it already has an open connection (hold that thought). Once connected, your session is available for use again:

PS> $s = Get-PSSession -Name FromW16AS01

PS> Invoke-Command -Session $s -ScriptBlock `

{Get-CimInstance Win32\_OperatingSystem}

SystemDirectory BuildNumber Version PSComputerName

--------------- ----------- ------- --------------

C:\Windows\system32 14393 10.0.14393 W16TGT01

<output truncated for brevity>

If a session is disconnected from its original host, you can connect to it from either the original host or another machine. After disconnecting the session FromW16DSC01 from its original host and testing available sessions on the local machine,

PS> Get-PSSession -ComputerName W16TGT01 |

Format-Table Id, Name, ComputerName, State,

Availability -AutoSize

Id Name ComputerName State Availability

-- ---- ------------ ----- ------------

3 FromW16AS01 W16TGT01 Opened Available

6 FromW16DSC01 W16TGT01 Disconnected None

you can see that the session FromW16DSC01 is disconnected and availability is shown as None. Connect to it in a similar way as before:

PS> Connect-PSSession -Name FromW16DSC01 -ComputerName W16TGT01

PS> Get-PSSession -ComputerName W16TGT01 |

Format-Table Id, Name, ComputerName, State,

Availability -AutoSize

Id Name ComputerName State Availability

-- ---- ------------ ----- ------------

3 FromW16AS01 W16TGT01 Opened Available

7 FromW16DSC01 W16TGT01 Opened Available

Disconnected sessions created by you on the local or other machine can be reconnected and used as shown. You can even connect to disconnected sessions created by other people as long as you have the credential details they used to create the session originally.

You can also use a PowerShell remoting session for copying files to and from a remote machine.

### 11.3.5 Copying Files across a PowerShell Remoting Session

PowerShell remoting is used to run commands on remote machines, as you saw in earlier sections, and have the results returned to you. In PowerShell v2–v4 you couldn’t copy files using a PowerShell remoting session. This changed in PowerShell v5 with the introduction of the -FromSession and -ToSession parameters on the Copy-Item cmdlet. Both of these new parameters take a single PSSession object as input.

This concept is best described by an example. Start by creating remoting sessions to two machines:

PS> $s1 = New-PSSession -ComputerName W16TGT01

PS> $s2 = New-PSSession -ComputerName W16DSC02

Now create a file on a remote machine:

PS> Invoke-Command -Session $s1 -ScriptBlock {

Get-Process | Out-File -FilePath c:\scripts\proc.txt}

You can copy the file from the remote machine to the local machine:

PS> Copy-Item -Path c:\scripts\proc.txt -FromSession $s1

Check that it arrived and then copy it to the second machine:

PS> Copy-Item -Path proc.txt -Destination C:\Scripts\ -ToSession $s2

A simple check confirms that the copy occurred:

PS> Invoke-Command -Session $s2 `

-ScriptBlock {Get-ChildItem -Path C:\Scripts\}

Everyone looks at the sequence of commands and thinks we can combine the copy steps:

PS> Copy-Item -Path c:\scripts\proc.txt -Destination C:\Scripts\ `

-FromSession $s1 -ToSession $s2

Copy-Item : '-FromSession' and '-ToSession' are mutually exclusive and cannot

be specified at the same time.

At line:1 char:1

+ Copy-Item -Path c:\scripts\proc.txt -Destination C:\Scripts\ -FromSe ...

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidArgument: (Microsoft.Power...namicParameters:CopyItemDynamicParameters) [Copy-Item],

ArgumentException

+ FullyQualifiedErrorId : InvalidInput,Microsoft.PowerShell.Commands.CopyItemCommand

Unfortunately, we can’t. The -FromSession and -ToSession parameters are mutually exclusive.

#### Note

This isn’t obvious from the help file because the parameters are shown in the same parameter set and their mutual exclusivity isn’t mentioned in the text.

You can copy multiple files across a PowerShell remoting session using wildcards to define the files.

## 11.4 Implicit Remoting

When doing non-interactive remoting, you have to call Invoke-Command every time you want to execute a remote operation. You can avoid this task by using Enter-PSSession to set up a remote interactive session. This approach makes remote execution easy but at the cost of making local operations difficult. In this section, we’ll look at a mechanism that makes both local and remote command execution easy. This mechanism is called implicit remoting.

#### Note

For implicit remoting to work, the execution policy on the client machine has to be configured to allow scripts to run, typically by setting it to RemoteSigned. This is necessary because implicit remoting generates a temporary module, and PowerShell must be allowed to execute scripts in order to load this module. If execution policy is set to Restricted or AllSigned, it won’t be able to do this. This requirement applies only to the local client machine. A remote server can still use a more restrictive policy. See [section 7.1.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-71-60.xhtml#ch07lev2sec1) for more information about execution policy.

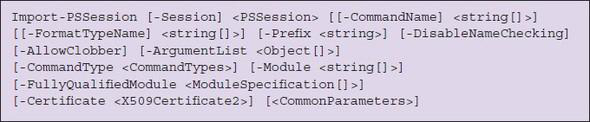
The goals of implicit remoting are to make the fact that remote operations are occurring invisible to the user and to have all operations look as much like local operations as possible. You can accomplish this goal by generating local proxy functions that run the remote commands under the covers. The user calls the local proxy, which takes care of the details involved in making the remote command invocation.

The net effect is that everything looks like a local operation because everything is a local operation.

### 11.4.1 Using Implicit Remoting

To set up the remote proxy functions mentioned in the previous section, use the Import-PSSession cmdlet. The syntax for this cmdlet is shown in [figure 11.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-114-97.xhtml#ch11fig07).

Figure 11.7: The syntax for the Import-PSSession cmdlet. This cmdlet is used to create local proxy commands that invoke the corresponding remote command on the target computer



Let’s explore how this cmdlet works by walking through an example. You’ll create a PSSession and then define a function in that session. The goal is to be able to execute this remote function as though it were defined locally. You want to implicitly remote the function. To do that, you call Import-PSSession, which generates a function that you can call locally. This local function does the remote call on your behalf—it acts as your proxy.

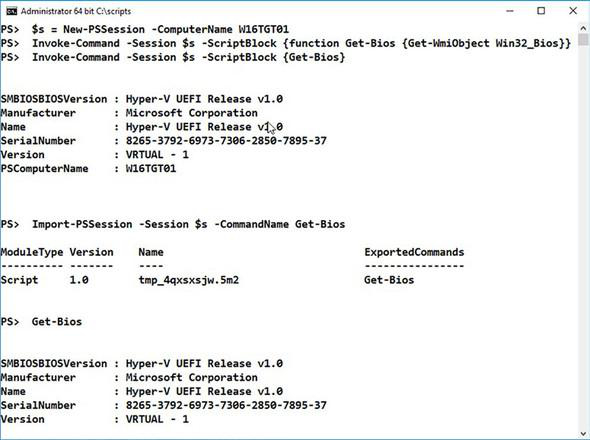
You’ll begin by creating the connection to a remote machine. You may need to get credentials for the remote host.

#### Note

In a domain environment, this step is unnecessary as long as your user account has sufficient privileges to access the remote endpoint. But if you want to log on as a different user, credentials will be required.

Establish a session on the remote machine, using credentials if necessary, as shown in [figure 11.8](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-114-97.xhtml#ch11fig08).

Figure 11.8: Example of implicit remoting



Next, you’ll use Invoke-Command to define a new function on the remote machine. This is the command you’ll import:

PS> Invoke-Command -Session $s -ScriptBlock {

function Get-Bios {Get-WmiObject Win32\_Bios}}

The new remote function, called Get-Bios, uses Windows Management Instrumentation (WMI) to retrieve information about the BIOS on the remote machine. Invoke this function through explicit remoting using Invoke-Command so you can see that it returns a set of information about the BIOS on the remote machine. Now use Import-PSSession to create a local proxy for this command:

PS> Import-PSSession -Session $s -CommandName Get-Bios

ModuleType Version Name ExportedCommands

---------- ------- ---- ----------------

Script 1.0 tmp\_4qxsxsjw.5m2 Get-Bios

You might recognize the output from this command—it’s the same thing you see when you do Get-Module. You now have a local Get-Bios command. Try running it:

PS> Get-Bios

SMBIOSBIOSVersion : Hyper-V UEFI Release v1.0

Manufacturer : Microsoft Corporation

Name : Hyper-V UEFI Release v1.0

SerialNumber : 8265-3792-6973-7306-2850-7895-37

Version : VRTUAL - 1

You get the same result you saw when you did the explicit remote invocation but without having to do any extra work to access the remote machine. The proxy command did that for you. This is the goal of implicit remoting: to make the fact that the command is being executed remotely invisible.

#### Note

This is a useful technique because you need to import the Exchange management module into your session if you’re administering an Exchange server over a PowerShell remoting session.

Let’s see how it all works.

### 11.4.2 How Implicit Remoting Works

When the user requests that a command be imported, a message is sent to the remote computer for processing. The import request processor looks up the command and retrieves the metadata (the CommandInfo object) for that command. That metadata is processed to simplify it, removing things like complex type attributes. Only the core remoting types are passed along. This metadata is received by the local machine’s proxy function generator. It uses this metadata to generate a function that will implicitly call the remote command.

Let’s take a closer look at what the generated proxy looks like. You can see the imported Get-Bios command using Get-Command:

PS> Get-Command Get-Bios

CommandType Name Version Source

----------- ---- ------- ------

Function Get-Bios 1.0 tmp\_4qxsxsjw.5m2

The output shows that you have a local function called Get-Bios. You can look at the definition of that function by using the Definition property on the CommandInfo object returned by Get-Command.

#### Listing 11.2: Definition of the Get-Bios proxy function

param(

[switch]${AsJob}

)

Begin {

try {

$positionalArguments =

& $script:NewObject collections.arraylist

foreach ($parameterName in

$PSBoundParameters.BoundPositionally)

{

$null = $positionalArguments.Add(

$PSBoundParameters[$parameterName] )

$null = $PSBoundParameters.Remove($parameterName)

}

$positionalArguments.AddRange($args)

$clientSideParameters =

Get-PSImplicitRemotingClientSideParameters`

$PSBoundParameters $False

$scriptCmd = { & $script:InvokeCommand `

@clientSideParameters `

-HideComputerName `

-Session (Get-PSImplicitRemotingSession `

-CommandName 'Get-Bios') `

-Arg ('Get-Bios', $PSBoundParameters,

$positionalArguments) `

-Script { param($name, $boundParams,

$unboundParams) & $name @boundParams

@unboundParams }`

}

$steppablePipeline =

$scriptCmd.GetSteppablePipeline($myInvocation.CommandOrigin)

$steppablePipeline.Begin($myInvocation.ExpectingInput,

$ExecutionContext)

} catch {

throw

}

}

Process {

try {

$steppablePipeline.Process($\_)

} catch {

throw

}

}

End {

try {

$steppablePipeline.End()

} catch {

throw

}

}

# .ForwardHelpTargetName Get-Bios

# .ForwardHelpCategory Function

# .RemoteHelpRunspace PSSession

Even though this output has been reformatted a bit to make it more readable, it’s a pretty complex function and uses many of the more sophisticated features covered in previous chapters. It uses advanced functions, splatting, scriptblocks, and steppable pipelines. Fortunately, you never have to write these functions yourself.

#### Note

You don’t have to create proxy functions for this particular scenario, but in [section 11.5.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-115-98.xhtml#ch11lev2sec18) you saw how this technique can be powerful in extending the PowerShell environment.

The Import-PSSession cmdlet does this for you. It will create a proxy function for each command it’s importing, which could lead to many commands. As well as generating proxy functions on your behalf, Import-PSSession creates a module to contain these functions.

The module name and path are temporary generated names. This module also defines an OnRemove handler (see [chapter 9](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-9-73.xhtml#ch09)) to clean up when the module is removed. To see the contents of the module, you can look at the temporary file that was created by using the module’s Path property:

PS> Get-Content (Get-Command Get-Bios).Module.Path

Alternatively, you can save the session to an explicitly named module for reuse with Export-PSSession. You’ll save this session as a module called bios:

PS> Export-PSSession -OutputModule bios -Session $s `

-type function -CommandName Get-Bios -AllowClobber

Directory: C:\Users\Richard\Documents\WindowsPowerShell\Modules\bios

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 08/05/2017 11:51 99 bios.format.ps1xml

-a---- 08/05/2017 11:51 528 bios.psd1

-a---- 08/05/2017 11:51 11627 bios.psm1

Executing this command creates a new module in your user module directory. It creates the script module file (.psm1), the module manifest (.psd1), and a file containing formatting information for the command. You use the -AllowClobber parameter because the export is using the remote session to gather the data. If it finds a command being exported that already exists in the caller’s environment, that would be an error. Because Get-Bios already exists, you have to use -AllowClobber.

Import the module into a new PowerShell session—remember to open it with elevated privileges:

PS> Import-Module bios

It returns right away. It can do this because it hasn’t set up the remote connection yet. This will happen the first time you access one of the functions in the module. Run Get-Bios:

PS> Get-Bios

Creating a new session for implicit remoting of "Get-Bios" command...

The term 'Get-Bios' 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.

+ CategoryInfo : ObjectNotFound: (Get-Bios:String) [], CommandNotFoundException

+ FullyQualifiedErrorId : CommandNotFoundException

+ PSComputerName : W16TGT01

When you run this command, you see a message indicating that a new connection is being created. But then you get an error saying the command Get-Bios isn’t found. That’s because you’re dynamically adding the function to the remote session. When you establish a new session, because you’re not adding the function, it isn’t there. In the next section, we’ll describe how to create remote endpoints that always contain your custom functions. There are a few other issues you need to be aware of when running commands remotely. We’ll look at those next.

## 11.5 Considerations When Running Commands Remotely

When you run commands on multiple computers, you need to be aware, at least to some extent, of how the execution environment can differ on the target machines. For example, the target machine may be running a different version of the operating system or it may have a different processor. There may also be differences in which applications are installed, how files are arranged, or where things are placed in the registry. In this section, we’ll look at a number of these issues. Don’t be put off by these issues—they’re not meant to scare you. They’re edge cases you need to be aware of to get the most out of PowerShell remoting.

### 11.5.1 Remote Session Startup Directory

When a user connects to a remote computer, the system sets the startup directory for the remote session to a specific value. This value will change depending on the version of the operating system on the target machine. If the machine is running Windows Vista, Windows Server 2003 R2, or a later version of Windows, the default starting location for the session is the user’s home directory, which is typically C:\Users\<UserName>.

On Windows Server 2003, the user’s home directory is also used: C:\Documents\Settings\<UserName>. For Windows XP, the default user’s home directory is used: C:\Documents\Settings\Default User.

#### Note

Windows Server 2003 and Windows XP are no longer supported by Microsoft and so should be less likely to be found in use with time. But from experience we can say that unsupported operating systems can easily linger for 10 years or more because of a special application that has to run on a particular version of Windows.

The default starting location can be obtained from either the $ENV:HOMEPATH environment or the PowerShell $HOME variable. By using these variables instead of hardcoded paths in your scripts, you can avoid problems related to these differences.

### 11.5.2 Profiles and Remoting

Most PowerShell users eventually create a custom startup script or profile that they use to customize their environment. These customizations typically include defining convenience functions and aliases. Although profiles are a great feature for customizing local interactive sessions, if the convenience commands they define are used in scripts that you want to run remotely, you’ll encounter problems. That’s because your profiles aren’t run automatically in remote sessions, and that means the convenience commands defined in the profile aren’t available in the remote session. In fact, the $PROFILE variable, which points to the profile file, isn’t even populated for remote sessions.

As a best practice, for production scripting you should make sure your scripts never become contaminated with elements defined by your profiles. One way to test this is to run the script from PowerShell.exe with the -NoProfile option, which looks like this:

powershell -NoProfile -File myscript.ps1

This command will run the script without loading your profile. If the script depends on anything defined in the profile, it will generate errors.

But for remote interactive sessions, it’d be nice to have the same environment everywhere. You can accomplish this by using Invoke-Command with the -FilePath parameter to send your profile file to the remote machine and execute it there. The set of commands you need to accomplish this are:

PS> $c = Get-Credential

PS> $s = New-PSSession -Credential $c -ComputerName targetComputer

PS> Invoke-Command -Session $s -FilePath $PROFILE

PS> Enter-PSSession $s

First, you get the credential for the target machine (this typically won’t be needed in the domain environment). Next, you create a persistent session to the remote computer. Then you use -FilePath on Invoke-Command to execute the profile file in the remote session. With the session properly configured, you can call Enter-PSSession to start your remote interactive session with all your normal customizations.

Alternatively, sometimes you may want to run a profile on the remote machine instead of your local profile. Because $PROFILE isn’t populated in your remote session, you’ll need to be clever to make this work. The key is that although $PROFILE isn’t set, $HOME is. You can use this to compose a path to your profile on the remote computer. The revised list of commands looks like this:

PS> $c = Get-Credential

PS> $s = New-PSSession -Credential $ -ComputerName targetComputer

PS> Invoke-Command -Session $s {

. "$home\Documents\WindowsPowerShell\profile.ps1" }

PS> Enter-PSSession $s

This command dot-sources (see [section 7.1.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-71-60.xhtml#ch07lev2sec4)) the profile file in the user’s directory on the remote machine into the session.

#### Note

This script won’t work on XP or Windows Server 2003. Change the script to use "$home\Documents and Setting\WindowsPowerShell\profile.ps1" as the profile path.

In this section, you learned how to cause your profile to be used to configure the remote session environment. Next, we’ll examine another area where these variations can cause problems.

### 11.5.3 Issues Running Executables Remotely

PowerShell remoting allows you to execute the same types of commands remotely as you can locally, including external applications or executables. The ability to remotely execute commands like shutdown to restart a remote host or ipconfig to get network settings is critical for system management.

For the most part, console-based commands will work properly because they read and write only to the standard input, output, and error pipes. Commands that won’t work are ones that directly call the Windows Console APIs, like console-based editors or text-based menu programs. The reason is that no console object is available in the remote session. Because these applications are rarely used any longer, this fact typically won’t have a big impact. But there are some surprises. For example, the net command will work fine most of the time, but if you do something like this (which prompts for a password)

PS> net use p: '\\machine1\c$' /user:machine1\user1 \*

Type the password for \\machine1\c$:

in a remote session, you’ll get an error:

[machine1]: > net use p: '\\machine1\c$' /user:machine1\user1 \*

net.exe : System error 86 has occurred.

+ CategoryInfo : NotSpecified: (System error 86 has

occurred.:String) [], RemoteException

+ FullyQualifiedErrorId : NativeCommandError

The specified network password is not correct.

Type the password for \\machine1\c$:

[machine1]: >

This command prompts for a password and returns an empty string.

The other kind of program that won’t work properly is commands that try to open a user interface (also known as “try to pop GUI”) on the remote computer. The program starts, but no window will appear. If the command eventually completes, control will be returned to the caller and things will be more or less fine. But if the process is blocked while waiting for the user to provide some input to the invisible GUI, the command will hang and you must stop it manually by pressing Ctrl-C. If the keypress doesn’t work, you’ll have to use some other mechanism to terminate the process.

One thing we can guarantee is that you’ll need to access files—but when you’re working remotely, how do you know which files you’re using?

### 11.5.4 Using Files and Scripts

When you enter an interactive PowerShell session and access a file, such as a script or text file, you’re obviously using the file on the remote machine. Remember that an interactive session is effectively like running a PowerShell session directly on the machine. But what about when you use Invoke-Command either directly or through a remoting session?

We’re going to be running a number of commands to the remote computer (W16TGT01), so we’ll create a remoting session:

PS> $s = New-PSSession -ComputerName W16TGT01

On the W16TGT01 machine, a file exists with these two lines:

Write-Host 'Run from W16TGT01'

Write-Host $env:COMPUTERNAME

You know that Invoke-Command is used to run commands through a remoting session:

PS> Invoke-Command -Session $s -ScriptBlock {C:\Scripts\PiA3e\FileTest.ps1}

Run from W16TGT01

W16TGT01

Sometimes you may have a script on your local machine that you need to run on remote machines. One solution would be to copy the script to the remote machines and run it as in the previous example. That would be inefficient if you’re dealing with hundreds or thousands of machines.

You can run a local script through a remoting session. Given a script on the local machine

Write-Host 'Run from W16AS01'

Write-Host $env:COMPUTERNAME

the –FilePath parameter is used to invoke a local script:

PS> Invoke-Command -Session $s -FilePath C:\Scripts\PiA3e\FileTest.ps1

Run from W16AS01

W16TGT01

Notice that the computer name that’s reported is the remote machine rather than the local machine, even though you’re running the script from your local disk.

One of the tenets of PowerShell remoting is isolation, but you can access local variables as well as local scripts.

### 11.5.5 Using Local Variables in Remote Sessions

When you use a variable in the scriptblock of a command sent to a remote machine, the assumption is that the variable is defined only in the session for the remote machine. For example, define a variable locally:

PS> $myvar = 123

Now, using the remoting session from the previous section (re-create a session if you closed that session), invoke a command using a variable with the same name:

PS> Invoke-Command -Session $s -ScriptBlock {"myvar is $myvar"}

myvar is

In the output of the command, you can see that the variable value was not made available in the remote session. In [chapters 6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-6-51.xhtml#ch06) and [7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-7-59.xhtml#ch07) we discussed scope modifiers and, for instance, how you can use variables from the global scope in your functions by prefixing them with $global:. PowerShell remoting provides a similar (but not identical) mechanism to allow you to use local variables in remote sessions, by using the $using: prefix. Let’s try the previous example again, but this time we’ll prefix the variable with $using:

PS> Invoke-Command -Session $s -ScriptBlock {"myvar is $using:myvar"}

Myvar is 123

Here’s what’s happening: By prefixing the variable name with $using (introduced in PowerShell v3), you’re telling PowerShell to copy the local value of the variable into the remote session. You’re using the local variable in the remote session. Where this differs from scope modifiers is that it’s one-way only. Changing the variable in the remote session won’t change the value of the local value. In fact, if you try to change the value of the $using variable in the remote session, you’ll get an error:

PS> Invoke-Command -Session $s { $using:myvar = 13 }

At line:1 char:30

+ invoke-command -localhost { $using:myvar = 13 }

+ ~~~~~~~~~~~~

The assignment expression is not valid. The input to an assignment operator

must be an object that is able to accept assignments, such as a variable

or a property.

+ CategoryInfo : ParserError: (:) [], ParentContainsErrorRecordException

+ FullyQualifiedErrorId : InvalidLeftHandSide

Now let’s look at more areas where accessing the console can cause problems and how to avoid these problems.

### 11.5.6 Reading and Writing to the Console

As you saw in the previous section, executables that read and write directly to the console won’t work properly. The same considerations apply to scripts that do things like call the System.Console APIs directly themselves. For example, call the [Console]::WriteLine() and [Console]::ReadLine() APIs in a remote session:

[machine1]: > [Console]::WriteLine('hi')

[machine1]: >

[machine1]: > [Console]::ReadLine()

[machine1]: >

Neither of these calls works properly. When you call the [Console]::WriteLine() API, nothing is displayed, and when you call the [Console]::ReadLine() API, it returns immediately instead of waiting for input.

It’s still possible to write interactive scripts, but you have to use the PowerShell host cmdlets and APIs:

[machine1]: > Write-Host Hi

Hi

[machine1]: >

[machine1]: > Read-Host "Input"

Input: some input

some input

If you use these cmdlets as shown in the example, you can read and write to and from the host, and the remoting subsystem will take care of making everything work.

With console and GUI issues out of the way, let’s explore how remoting affects the objects you’re passing back and forth.

### 11.5.7 Remote Output vs. Local Output

Much of the power in PowerShell comes from the fact that it passes around objects instead of strings. In this section, you’ll learn how remoting affects these objects.

When PowerShell commands are run locally, you’re working directly with the live .NET objects, which means that you can use the properties and methods on these objects to manipulate the underlying system state. The same isn’t true when you’re working with remote objects. Remote objects are serialized—converted into a form that can be passed over the remote connection—when they’re transmitted between the client and the server, and deserialized when received by the client machine.

#### Note

The biggest difference you’ll find is that the objects returned from a remoting session don’t have any of the methods you’d have available from the same object generated locally.

Typically, you can use deserialized objects as you’d use live objects, but you must be aware of their limitations. Another thing to be aware of is that the objects that are returned through remoting will have had properties added that allow you to determine the origin of the command.

#### PowerShell Serialization

Because you can’t guarantee that every computer has the same set of types, the PowerShell team chose to limit the number of types that serialize with fidelity, where the remote type is the same type as the local type and the object is fully re-created at the receiving end. To address the restrictions of a bounded set of types, types that aren’t serialized with fidelity are serialized as a collection of properties, also called a property bag. This property bag has a special property, TypeNames, which records the name of the original type. The serialization code takes each object and adds all its properties to the property bag. Recursively, it looks at values of each the members. If the member value isn’t one of the ones supported with fidelity, a new property bag is created, with members of the member’s values added to it, and so on. This approach preserves structure if not the type and allows remoting to work uniformly everywhere.

#### Default Serialization Depth

The approach we have described allows any object to be encoded and transferred to another system. But there’s another thing to consider: objects have members that contain objects that contain members, and so on. The full tree of objects and members can be complex. Transferring all the data makes the system unmanageably slow. This is addressed by introducing the idea of serialization depth. The recursive encoding of members stops when this serialization depth is reached. The default for objects is 1.

The final source of issues when writing portable, remotable scripts has to do with processor architectures and the operating system differences they entail. We’ll work through this final set of issues in the next section of this chapter.

### 11.5.8 Processor Architecture Issues

The last potential source of problems we’ll explore is the fact that the target machine may be running on a different processor architecture (64-bit versus 32-bit) than the local machine. If the remote computer is running a 64-bit version of Windows and the remote command is targeting a 32-bit session configuration, such as Microsoft.PowerShell32, the remoting infrastructure loads a Windows 32-bit process on a Windows 64-bit (WOW64) process, and Windows automatically redirects all references to the $ENV:Windir\System32 directory to the $ENV:WINDIR\SysWOW64 directory. For the most part, everything will still work (that’s the point of the redirection), unless you try to invoke an executable in the System32 directory that doesn’t have a corresponding equivalent in the SysWOW64 directory.

To find the processor architecture for the session, you can check the value of the $ENV:PROCESSOR\_ARCHITECTURE variable. The following command finds the processor architecture of the session in the $s variable. Try this first with the 32-bit configuration:

PS> Invoke-Command -ConfigurationName microsoft.powershell32 `

-ComputerName localhost { $ENV:PROCESSOR\_ARCHITECTURE }

x86

You get the expected x86 result, indicating a 32-bit session, and on the 64-bit configuration

PS> Invoke-Command -ConfigurationName microsoft.powershell `

-ComputerName localhost { $ENV:PROCESSOR\_ARCHITECTURE }

AMD64

you get AMD64, indicating a 64-bit configuration.

This is the last remoting consideration we’re going to look at in this chapter. Don’t let these issues scare you—remember, they’re mostly edge cases. With some attention to detail, the typical script should have no problems working as well remotely as it does locally. The PowerShell remoting system goes to great lengths to facilitate a seamless remote execution experience. But it’s always better to have a heads-up on some of the issues so you’ll know where to start looking if you run into a problem.

Up to now we’ve been using the default remoting configuration. In the next section, we’ll look at how you can create and configure your own specialized remoting configuration.

## 11.6 Building Custom Remoting Services

So far, we’ve looked at remoting from the service consumer perspective. It’s time for you to take on the role of service creator instead.

The most common remoting scenario for administrators is the one-to-many configuration, in which one client computer connects to a number of remote machines in order to execute remote commands on those machines. This is called the fan-out scenario because the connections fan out from a single point, and this is what you’ve been using in the previous sections.

In enterprises and hosted solution scenarios, you’ll find the opposite configuration, where many client computers connect to a single remote computer, such as a file server or a kiosk. This many-to-one arrangement is known as the fan-in configuration. This mechanism is used when remote connecting to Exchange servers or Active Directory domain controllers.

Windows PowerShell remoting supports both fan-out and fan-in configurations. In the fan-out configuration, PowerShell remoting connects to the remote machine using the WinRM service running on the target machine. When the client connects to the remote computer, the WS-MAN protocol is used to establish a connection to the WinRM service. The WinRM service then launches a new process (wsmprovhost.exe) that loads a plug-in that hosts the PowerShell engine.

#### PowerShell remoting protocols

The transport mechanism used in PowerShell remoting consists of a five-layer stack. The stack (from top to bottom) consists of the following:

* The PowerShell Remoting Protocol (MS-PSRP)—<https://msdn.microsoft.com/en-us/library/dd357801.aspx>
* WS-MAN (implemented by the WinRM service)—<http://mng.bz/DB74> and <https://msdn.microsoft.com/en-us/library/cc251395.aspx>.
* Simple Object Access Protocol (SOAP)—Provides an XML-based messaging framework
* HTTP and HTTPS
* TCP/IP

Creating a new process for each session is fine if there aren’t many users connecting to the service. But if several connections are expected, as is the case for a high-volume service, the one-process-per-user model won’t scale well. To address this issue, an alternate hosting model, targeted at developers, is available for building custom fan-in applications on top of PowerShell remoting. Instead of using the WinRM service to host WS-MAN and the PowerShell plug-in, Internet Information Services (IIS) is used. In this model, instead of starting each user session in a separate process, all the PowerShell sessions are run in the same process along with the WS-MAN protocol engine.

Having all the sessions running in the same process has certain implications. Because PowerShell lets you get at pretty much everything in a process, multiple users running unrestricted in the same process could interfere with one another. On the other hand, because the host process persists across multiple connections, it’s possible to share process-wide resources like database connections between sessions.

Given the lack of session isolation, this approach isn’t intended for full-featured general-purpose PowerShell remoting. Instead, it’s designed for use with constrained, special-purpose applications using PowerShell remoting. To build these applications, you need two things:

* A way to create a constrained application environment
* A way to connect to PowerShell remoting so the user gets the environment you’ve created instead of the default PowerShell configuration

We’ll start with the second one first and look at how you specify custom remoting endpoints.

### 11.6.1 Working with Custom Configurations

When connecting to a computer by name through PowerShell remoting, the remoting infrastructure will always connect to the default PowerShell remoting service. In the non-default connection case, you also have to specify the configuration on the target computer to connect to. A configuration is made up of three elements:

* The name you use to connect to the endpoint
* A script that will be run to configure the sessions that will run in the endpoint
* An ACL used to control who has access to the endpoint

When using the Invoke-Command, New-PSSession, or Enter-PSSession cmdlets, you can use the -ConfigurationName parameter to specify the name of the session configuration you want to connect to. Alternatively, you can override the normal default configuration by setting the $PSSessionConfigurationName preference variable to the name of the endpoint you want to connect to.

When you connect to the named endpoint, a PowerShell session will be created, and then the configuration script associated with the endpoint will be executed. This configuration script should define the set of capabilities available when connecting to that endpoint. For example, there may be different endpoints for different types of management tasks—managing a mail server, managing a database server, or managing a web server. For each task, a specific endpoint would be configured to expose the appropriate commands (and constraints) required for performing that task.

### 11.6.2 Creating a Custom Configuration

Continuing our theme of remote monitoring from [section 11.2.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-112-95.xhtml#ch11lev2sec9), let’s create a configuration that exposes a single custom command, Get-PageFaultRate. This command will return the page fault rate from the target computer.

#### Session Configuration

Every remoting connection will use one of the named configurations on the remote computer. These configurations set up the environment for the session and determine the set of commands visible to users of that session.

When remoting is initially enabled, a default configuration is created on the system called Microsoft.PowerShell (on 64-bit operating systems, there’s also the Microsoft.PowerShell32 endpoint). This endpoint is configured to load the default PowerShell configuration with all commands enabled. The security descriptor for this configuration is set so that only members of the local Administrators group can access the endpoint.

You can use the session configuration cmdlets to modify these default session configurations, to create new session configurations, and to change the security descriptors of all the session configurations. These cmdlets are shown in [table 11.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11table04).

| **Table 11.4: The cmdlets for managing the remoting endpoint configurations** | |
| --- | --- |
| **Cmdlet** | **Description** |
| Disable-PSSessionConfiguration | Denies access to the specified session configuration on the local computer by adding an “Everyone AccessDenied” entry to the access control list (ACL) on the configuration |
| Enable-PSSessionConfiguration | Enables existing session configurations on the local computer to be accessed remotely |
| Get-PSSessionConfiguration | Gets a list of the existing, registered session configurations on the computer |
| Register-PSSessionConfiguration | Creates and registers a new session configuration |
| Set-PSSessionConfiguration | Changes the properties of an existing session configuration |
| Unregister-PSSessionConfiguration | Deletes the specified registered session configurations from the computer |
| New-PSSessionConfigurationFile | Creates a PowerShell data language file (see module manifests) with a .pssc extension that defines a session configuration |
| Test-PSSessionConfigurationFile | Validates the contents of a session configuration file, verifying that the keys and values in the file are all valid (introduced in PowerShell v4). |

#### Registering the Endpoint Configuration

Endpoints are created using the Register-PSSessionConfiguration cmdlet and are customized by registering a startup script. In this example, you’ll use a simple startup script that defines a single function, Get-PageFaultRate. The script looks like this:

PS> @'

function Get-PageFaultRate {

(Get-WmiObject Win32\_PerfRawData\_PerfOS\_Memory).PageFaultsPersec

}

'@ > Initialize-HMConfiguration.ps1

Before you can use this function, you need to register the configuration, specifying the full path to the startup script. Call this new configuration wpia1. From an elevated PowerShell session, run the following command to create the endpoint:

PS> Register-PSSessionConfiguration -Name wpia1 `

-StartupScript $pwd/Initialize-HMConfiguration.ps1 -Force

WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin

Type Keys Name

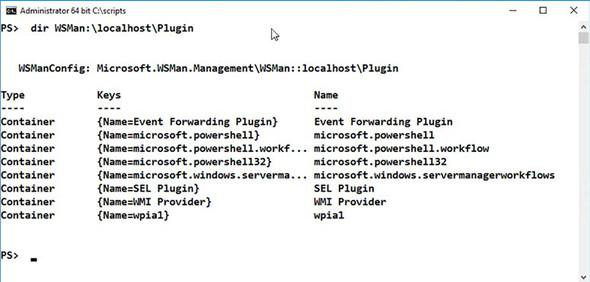
---- ---- ----

Container {Name=wpia1} wpia1

The output of the command shows that you’ve created an endpoint in the WSMan plug-in folder. To confirm this use (see [figure 11.9](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11fig09)), run the following:

PS> dir wsman:\localhost\plugin

Figure 11.9: Remoting endpoints including the newly created wpia1



This shows a list of all the existing endpoints, including the one you created, wpia1. Now test this endpoint with Invoke-Command and run the function defined by the startup script:

PS> Invoke-Command localhost -ConfigurationName wpia1 {

Get-PageFaultRate }

68200956

This code verifies that the endpoint exists and is properly configured. Now clean up by unregistering the endpoint:

PS> Unregister-PSSessionConfiguration -Name wpia1 -Force

Rerun the dir command in [figure 11.9](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11fig09) to verify that the endpoint has been removed.

This covers the basic tasks needed to create a custom PowerShell remoting endpoint using a configuration script to add additional functionality to the session defaults. Our ultimate goal, though, is to create a custom endpoint with reduced functionality, exposing a restricted set of commands to qualified users, so clearly, we aren’t finished yet. There are two remaining pieces to look at: controlling individual command visibility, which we’ll get to in a while, and controlling overall access to the endpoint, our next topic.

### 11.6.3 Access Controls and Endpoints

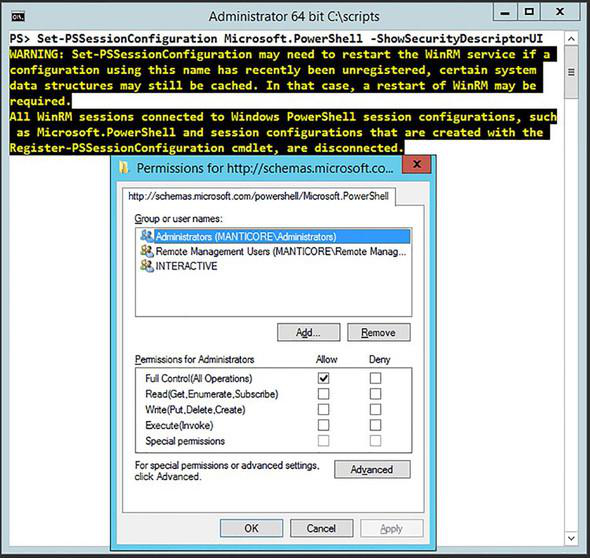
By default, only members of the Administrators group on a computer have permission to use the default session configurations. To allow users who aren’t part of the Administrators group to connect to the local computer, you have to give those users Execute permissions on the session configurations for the desired endpoint on the target computer. For example, if you want to enable non-administrators to connect to the default remoting Microsoft.PowerShell endpoint, you can do so by running the following command:

PS> Set-PSSessionConfiguration Microsoft.PowerShell `

-ShowSecurityDescriptorUI

This code launches the dialog box shown in [figure 11.10](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11fig10).

Figure 11.10: This dialog box is used to enable the Execute permission on the default remoting configuration. Use this dialog box to allow a user who isn’t a member of the Administrators group to connect to this computer using PowerShell remoting



You add the name of a user or a group you want to enable to the list, then select the Execute (Invoke) check box. Then dismiss the dialog box by clicking OK. At this point, you’ll get a prompt telling you that you need to restart the WinRM service for the change to take effect. Do so by running Restart-Service winrm as shown here:

PS> Restart-Service winrm

Once the service is restarted, the user or group you’ve enabled can connect to the machine using remoting.

#### Setting Security Descriptors on Configurations

When Enable-PSRemoting creates the default session configuration, it doesn’t create explicit security descriptors for the configurations. Instead, the configurations inherit the security descriptor of the RootSDDL. The RootSDDL is the security descriptor that controls remote access to the listener, which is secure by default. To see the RootSDDL security descriptor, run the Get-Item command as shown:

PS> Get-Item wsman:\localhost\Service\RootSDDL

WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Service

Type Name SourceOfValue Value

---- ---- ------------- -----

System.String RootSDDL O:NSG:BAD:P(A;;GA;;;BA)

(A;;GR;;;IU)

S:P(AU;FA;GA;;;WD)

AU;SA;GXGW;;;WD)

The string format shown in the Value output in the example uses the syntax defined by the Security Descriptor Definition Language (SDDL), which is documented in the Windows Data Types specification MS-DTYP in [section 2.5.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-25-21.xhtml#ch02lev2sec16) at <http://mng.bz/QpKC>.

To change the RootSDDL, use the Set-Item cmdlet in the WSMan: drive. To change the security descriptor for an existing session configuration, use the Set-PSSessionConfiguration cmdlet with the -SecurityDescriptorSDDL or -ShowSecurityDescriptorUI parameter.

At this point, you know how to create and configure an endpoint and how to control who has access to that endpoint. But in your configuration, all you’ve done is add new commands to the set of commands you got by default. You haven’t addressed the requirement to constrain the environment.

### 11.6.4 Constraining a PowerShell Session

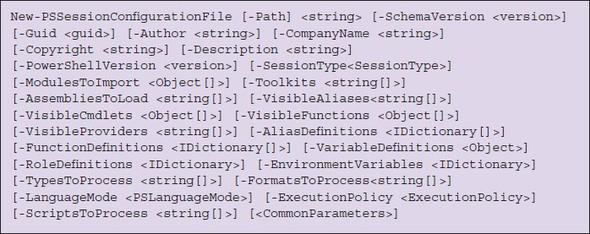
In [section 11.6.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11lev2sec26) you saw how to create a new remoting endpoint using Register -PSSessionConfiguration, and in the previous section you saw how to control who can access a particular endpoint. In this section, you’ll learn how to control, or constrain, what can be done through a particular endpoint. This involves limiting the variables and commands available to the user of the session. You accomplish this by controlling command and variable visibility. You’re creating a constrained endpoint.

The idea behind a constrained endpoint is that it allows you to provide controlled access to services on a server in a secure manner. This is the mechanism that the hosted Exchange product [Outlook.com](http://outlook.com/) uses to constrain who gets to manage which sets of mailboxes. The mechanism can also be used in PowerShell Web Access to control access to a server and the commands that can be run on that server.

In PowerShell v2 you had to create a complex script to configure a new endpoint. The script involved manipulating the visibility of cmdlets and variables plus the definition of any new functionality you required.

In PowerShell v3 and later this task became much simpler thanks to the introduction of the New-PSSessionConfigurationFile cmdlet; the syntax is shown in [figure 11.11](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11fig11).

Figure 11.11: New-PSSessionConfigurationFile syntax



The only required parameter is the path to the new configuration file:

PS> New-PSSessionConfigurationFile -Path .\Defaults.pssc

Configuration files are given a .pssc extension. The .pssc file structure is similar to a module manifest; it’s a big PowerShell hashtable with name-value pairs. If you examine defaults.pssc (see download) produced by the example, you’ll see that you can control a large number of configuration items, including these:

* Execution policy (controls which, if any, scripts can be run)
* Language mode
* Session type
* PowerShell version
* Existing aliases, cmdlets, functions, and providers that are visible in the endpoint
* New aliases, functions, and variables to create for the endpoint
* Format and type files to load and scripts to process

Language mode for a session configuration controls the types of things that can be executed in a session. The more secure you need the session to be, the more restrictive the language mode session should be. The options are shown in [table 11.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11table05).

| **Table 11.5: Remoting endpoint language options** | |
| --- | --- |
| **Option** | **Meaning** |
| FullLanguage | All PowerShell language elements are permitted. |
| ConstrainedLanguage | Commands that contain scripts to be evaluated are not allowed. User access is restricted to .NET framework types, objects, or methods. (This is the mode that PowerShell runs in on WinRT devices.) |
| RestrictedLanguage | Users may run cmdlets and functions. Scriptblocks aren’t allowed. Only the following variables are allowed: $PSCulture, $PSUICulture, $True, $False, and $Null. Basic comparison operators are allowed. Assignment statements, property references, and method calls aren’t permitted. (This is the language mode used in module manifests, sometimes also called data language mode because it can only describe data.) |
| NoLanguage | Users may run simple pipelines containing cmdlets and functions. No language elements such as scriptblocks, variables, or operators are permitted in the pipeline. |

As you progress down the table, the things you can do in the endpoint become more limited until Nolanguage, when you’re only allowed to run basic pipelines containing cmdlets and functions. The session capabilities are also controllable by restricting the list of cmdlets and functions available to a user. For example, you can restrict the functionality of an endpoint so that a user can only reset their password in Active Directory!

The session type works in conjunction with the language mode. The session type options are listed in [table 11.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11table06).

| **Table 11.6: Session options for remoting endpoints** | | |
| --- | --- | --- |
| **Option** | **Meaning** | **Default language mode** |
| Default | Adds the Microsoft.PowerShell.Core snap-in to the session. This includes the Import-Module and Add-PSSnapin cmdlets so users can import other modules and snap-ins unless you explicitly prohibit the use of the cmdlets. | FullLanguage |
| RestrictedRemoteServer | Includes only the following proxy functions: Exit-PSSession, Get-Command, Get-FormatData, Get-Help, Measure-Object, Out-Default, and Select-Object. Use New -PSSessionConfigurationFile to add modules, functions, scripts, and other features to the session. | NoLanguage |
| Empty | No modules or snap-ins are added to the session by default. Use New-PSSessionConfigurationFile to add modules, functions, scripts, and other features to the session. This option is designed for you to create custom sessions by adding selected commands. If you don’t add commands to an empty session, the session is limited to expressions and might not be usable. | NoLanguage |

You can explicitly control the visibility of PowerShell elements using the –Visible\* parameters shown in [figure 11.11](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11fig11). This is a “white list” action. If a cmdlet or other element isn’t on the list, you won’t see it and therefore you won’t be able to use it directly.

#### Tip

When using the –Visible\* parameters, if you don’t want to make anything visible for a particular type of command, don’t use the parameter. A commented-out default value will be written to the .pssc file.

An example of an extremely constrained endpoint is provided in the following listing.

#### Listing 11.3: ComplexConstrainedConfiguration.ps1

New-PSSessionConfigurationFile `

-Path .\ComplexConstrainedConfiguration.pssc `

-Schema '1.0.0.0' `

-Author 'Richard' `

-Copyright '(c) PowerShell in Action Third Edition. All rights reserved.' `

-CompanyName 'PowerShell in Action' `

-Description 'Complex Constrained Configuration.' `

-ExecutionPolicy RemoteSigned `

-PowerShellVersion '5.0' `

-LanguageMode NoLanguage `

-SessionType RestrictedRemoteServer `

-FunctionDefinitions @{Name='Get-HealthModel';ScriptBlock={@{

Date = Get-Date

FreeSpace = (Get-PSDrive c).Free

PageFaults = (Get-WmiObject `

Win32\_PerfRawData\_PerfOS\_Memory).PageFaultsPersec

TopCPU = Get-Process | Sort-Object -Descending CPU

TopWS = Get-Process | Sort-Object -Descending WS

}};Options='None'} `

-VisibleProviders 'FileSystem','Function','Variable'

The execution policy is set to RemoteSigned, but in reality, you won’t be able to run scripts, as you’ll see in a while. Language mode is set to NoLanguage (see [table 11.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11table05)) and session type to RestrictedRemoteServer ([table 11.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11table06)). Three providers are made visible, but no modules, cmdlets, aliases, or variables are made available in the session.

A function to get the health of the system is defined and will be created when the endpoint is created. Run the script in [listing 11.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11ex03) to create a configuration file. The fidelity of a configuration file can be tested:

PS> Test-PSSessionConfigurationFile -Path `

.\ComplexConstrainedConfiguration.pssc -Verbose

True

In the event of an error in the file, you will see the error only if you use the –Verbose parameter:

PS> Test-PSSessionConfigurationFile -Path .\ErrorConfiguration.pssc `

-Verbose

VERBOSE: The member 'LanguageMode' must be a valid enumeration type "System.

Management.Automation.PSLanguageMode".

Valid enumeration values are "FullLanguage,RestrictedLanguage,NoLanguage,

ConstrainedLanguage". Change the member to the correct type in the file C:\

MyData\PowerShellinAction3e\Code\Chapter11\ErrorConfiguration.pssc.

False

Creating the endpoint is performed with Register-PSSessionConfiguration. In the following example, any existing instances of the endpoint are removed—a useful technique when testing:

PS> Unregister-PSSessionConfiguration -Name wpiaccs -Force

PS> Register-PSSessionConfiguration –Path ` .\

ComplexConstrainedConfiguration.pssc -Name wpiaccs -Force

WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin

Type Keys Name

---- ---- ----

Container {Name=wpiaccs} wpiaccs

You can see the new endpoint:

PS> dir WSMan:\localhost\Plugin\

WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin

Type Keys Name

---- ---- ----

Container {Name=Event Forwarding Plugin} Event Forwarding Plugin

Container {Name=microsoft.powershell} microsoft.powershell

Container {Name=microsoft.powershell.w... microsoft.powershell.workflow

Container {Name=microsoft.powershell32} microsoft.powershell32

Container {Name=microsoft.windows.serv... microsoft.windows.server...

Container {Name=SEL Plugin} SEL Plugin

Container {Name=WMI Provider} WMI Provider

Container {Name=wpiaccs} wpiaccs

A remoting session can be created to the new endpoint. Notice that you have to give the name of the configuration (endpoint) that you used when performing the registration:

PS> $s = New-PSSession -ComputerName localhost -ConfigurationName wpiaccs

The session can now be used as normal. Let’s start by checking the commands available:

PS> Invoke-Command -Session $s -ScriptBlock {Get-Command | select Name}

Name PSComputerName RunspaceId

---- -------------- ----------

Clear-Host localhost 0377a4f9-5924-4cb0-83f9-a87f8a335147

Exit-PSSession localhost 0377a4f9-5924-4cb0-83f9-a87f8a335147

Get-Command localhost 0377a4f9-5924-4cb0-83f9-a87f8a335147

Get-FormatData localhost 0377a4f9-5924-4cb0-83f9-a87f8a335147

Get-HealthModel localhost 0377a4f9-5924-4cb0-83f9-a87f8a335147

Get-Help localhost 0377a4f9-5924-4cb0-83f9-a87f8a335147

Measure-Object localhost 0377a4f9-5924-4cb0-83f9-a87f8a335147

Out-Default localhost 0377a4f9-5924-4cb0-83f9-a87f8a335147

Select-Object localhost 0377a4f9-5924-4cb0-83f9-a87f8a335147

#### Note

When you look at this list of commands, you may wonder why some of them are included. For example, Measure-Object seems like a strange thing to have on the list. The reason these commands are included is that they’re needed to implement some of the elements of the PowerShell Remoting Protocol. In particular, they’re used to help with the command-discovery component described in the PowerShell Remoting Protocol Specification (MS-PSRP) section 3.1.4.5, “Getting Command Metadata.”

Compare that with the results on the machine we’re using to test the code for this book:

PS> Get-Command | Measure-Object | select Count

Count

-----

2658

Our session is constrained! You’ll notice that the function we defined, Get-HealthModel, is in the list of commands. Let’s check that it works:

PS> Invoke-Command -Session $s -ScriptBlock {get-healthmodel}

Name Value

---- -----

Date 08/05/2017 12:57:29

TopWS {System.Diagnostics.Proces...

PageFaults 146394771

FreeSpace 67302338560

TopCPU {System.Diagnostics...

The observant reader will have noticed that we used Get-Date in the function, but it isn’t in the list of commands we obtained from Get-Command. Does this mean we can use it directly even though we didn’t explicitly make it visible in our configuration definition?

PS> Invoke-Command -Session $s -ScriptBlock {Get-Date}

The term 'Get-Date' 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.

And the answer is no! This is an important point to understand because it’s the key to creating a restricted special-purpose endpoint: an external call can only access visible commands, but these commands, because they’re defined as part of the configuration, can see all the other commands in the configuration. This means that an externally visible command can call any internal commands in the session. If the user makes an external call to a visible command, that visible command is able to call the private commands.

#### Note

All the error messages in this section will be truncated to show only the error text for brevity.

What about using it in a script block or function?

PS> Invoke-Command -Session $s -ScriptBlock { & {Get-Date}}

The syntax is not supported by this runspace. This can occur if the runspace is in no-language mode.

PS> Invoke-Command -Session $s -ScriptBlock {function MyGetDate { [string] (Get-Date) }; MyGetDate}

The syntax is not supported by this runspace. This can occur if the runspace is in no-language mode.

If you want to be able to create functions and scriptblocks, you need to be using FullLanguage mode in your endpoint. What about adding extra modules into the endpoint—modules provide extra functionality? Let’s see what modules you have available:

PS> Invoke-Command -Session $s -ScriptBlock {Get-Module -ListAvailable}

The term 'Get-Module' is not recognized as the name of a cmdlet, function,

script file, or operable program. Check thespelling of the name, or if a path

was included, verify that the path is correct and try again.

You can’t see any modules so you can’t load them because you don’t know what’s on the system. You might think about trying to import modules that you know are present, but it will fail. The endpoint is locked down to prevent any further functionality being imported. The function we defined as part of our configuration used variables. Can you use variables in your endpoint?

PS> Invoke-Command -Session $s -ScriptBlock {$x = 123; $x}

The syntax is not supported by this runspace. This can occur if the runspace

is in no-language mode.

No, they’re not allowed. There’s still a lot of functionality in legacy commands that you may think to use:

PS> Invoke-Command -Session $s -ScriptBlock {ping 127.0.0.1}

The term 'PING.EXE' 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.

Notice that the full name of the executable was recognized—but you’re not allowed to run it. The final piece of functionality you may try is to run a script. You can try a simple script testch11.ps1 consisting of

Get-Service | Sort-Object Status

Try this:

PS> Invoke-Command -Session $s -ScriptBlock {C:\TestScripts\testch11.ps1}

The term 'C:\TestScripts\testch11.ps1' 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.

Again, the endpoint won’t allow you to run anything beyond what it’s been told is allowed. You do have a constrained remoting session.

#### Note

The example we’ve used is extreme but was designed to illustrate that you can create an endpoint and control exactly what functionality is exposed.

Step back and think about what you’ve accomplished here. With a few lines of code, you’ve defined a secure remote service. From the users’ perspective, by using Import -PSSession they’re able to install the contents of the session to use the services you expose—by connecting to the service.

Constrained sessions combined with implicit remoting results in an extremely flexible system, allowing you to create precise service boundaries with little server-side code and no client code. Consider how much code would be required to create an equivalent service using alternate technologies!

We’ll close the chapter with a new remoting feature introduced with PowerShell v5.

## 11.7 PowerShell Direct

You normally use the computer name to define the remote machine for PowerShell remoting, whether you’re using an interactive session, a persistent session, or Invoke -Command in standalone mode (no persistent session). PowerShell v5.1 supplies some new options. You can use a Hyper-V virtual machine name (not necessarily the same as the computer name) or the virtual machine ID (a GUID).

The options to use a virtual machine name or ID apply only under these circumstances:

* The virtual machine must be running on the local host.
* You must be logged on to the Hyper-V host as a Hyper-V administrator.
* You must supply valid credentials for the virtual machine—not domain credentials.
* The host operating system must be Windows 10, Windows Server 2016, or later.
* The virtual machine operating system must be Windows 10, Windows Server 2016, or later.

You can use the virtual machine name or ID to connect, but it’s usually easier to use the name:

PS> Get-VM | where State -eq 'Running' |

select Name, Id

Name Id

---- --

W16AS01 2a1eabc2-e3cd-495c-a91f-51a1ad43104c

W16DSC01 867c8460-a4fb-4785-9b7c-f27c9351db3c

W16TGT01 be4a5a3f-fc20-49f9-bb0f-b575c85e5734

Create a credential for the administrator account on the remote machine and then use the virtual machine name to connect:

PS> $cred = Get-Credential -Credential W16TGT01\Administrator

PS> Invoke-Command -VMName W16TGT01 -ScriptBlock {Get-Process} `

-Credential $cred

Either of these options will also work:

PS> Invoke-Command -VMId be4a5a3f-fc20-49f9-bb0f-b575c85e5734 `

-ScriptBlock {Get-Process} -Credential $cred

PS> Invoke-Command -VMGuid be4a5a3f-fc20-49f9-bb0f-b575c85e5734 `

-ScriptBlock {Get-Process} -Credential $cred

**Note**

VMGuid is an alias for VMId.

You can create a persistent remoting session:

PS> $s = New-PSSession -VMName W16TGT01 -Credential $cred

PS> Invoke-Command -Session $s -ScriptBlock {Get-Process}

Or you can work interactively:

PS> Enter-PSSession -VMName W16TGT01 -Credential $cred

[W16TGT01]: PS C:\Users\Administrator\Documents>

Use Exit-PSSession to close the interactive session.

There are a few things you need to remember when using PowerShell Direct:

* It’s only for Hyper-V virtual machines.
* You can ignore network and firewall configurations; you’re connecting over the VM bus rather than the network.
* PowerShell must be run with elevated privileges.

And with this, we’ve come to end of our coverage of the remoting features in PowerShell.

## 11.8 Summary

* Many PowerShell commands have built-in remoting using a -ComputerName parameter.
* Cmdlets with built-in remoting use a variety of connectivity mechanisms including DCOM and RPC.
* Invoke-Command uses WS-MAN for remote connectivity.
* You can create an interactive remoting session with Enter-PSSession.
* Interactive remoting sessions are closed with Exit-PSSession.
* Windows Server 2012 and later enable remoting by default. Azure IAAS virtual machines running Server 2012 R2 or higher also enable PowerShell remoting by default.
* All client operating systems and Windows Server 2008 R2 and earlier need remoting enabled by running Enable-PSRemoting.
* Additional configuration may be required in a non-domain environment.
* Users are authenticated using Kerberos in a domain environment when creating remoting sessions.
* Other authentication mechanisms are available for non-domain scenarios.
* New-PSSession is used to create a persistent remoting session.
* Invoke-Command and interactive sessions can use an existing session created with New-PSSession.
* PowerShell sessions can be disconnected and later reconnected. The reconnection can happen on the machine on which the session was created or another machine.
* You can connect to a disconnected session created by another user if you have the correct credential information.
* Copy-Item has -FromSession and -ToSession parameters that enable you to copy files across PowerShell remoting sessions.
* Implicit remoting enables you to import functionality from the remote system into your session. You can save the imported commands as a module.
* Profiles don’t run by default in remoting sessions.
* Scripts on the local or remote machine can be run through a remoting session.
* Local variables can be accessed in a remoting session via the $using scope modifier.
* Custom endpoints can be created to constrain the functionality available to a user through a specific remoting connection.
* PowerShell Direct enables remoting over the VM bus from a Hyper-V host to a virtual machine on that host.

In the next chapter, we’ll look at a feature introduced in PowerShell v3: PowerShell workflows.

## Chapter 12: PowerShell Workflows

### Overview

This chapter covers

* Workflow overview and architecture
* Workflow keywords
* Workflow parameters
* Workflow cmdlets

*“Hi ho, hi ho. It’s off to work we go!”  
Snow White and the Seven Dwarfs*

At the beginning of every new release of PowerShell, a planning cycle takes place during which a number of major themes are identified for the release. During the planning cycle for PowerShell v3, one of the key themes was identified as multi-machine management—the ability to provision, manage, and monitor a large number of machines in a datacenter. Accomplishing this goal, however, would require changing not only PowerShell but also some related products such as Server Manager.

In Windows 7, which included PowerShell v2 and the first version of Server Manager, the focus was mainly on machine-to-machine management. Server Manager could attach to a remote machine but could manage only one machine at a time. PowerShell was a bit more sophisticated in that it could deal with multiple machines but only if you wanted to do the same thing to each machine.

The key element that was missing was orchestration. In an orchestra, the conductor doesn’t play any music. Instead, they direct the orchestra members, each of whom has a specialized role to play and a time to play that role (music). This direction of the orchestra is called, not surprisingly, orchestration. A central control point sequences the flow of work to the individual workers. It was clear that workflow management had to become part of the overall management stack to achieve large-scale multi-machine management.

PowerShell workflows were introduced in PowerShell v3. Workflows give you another option when deciding how you’ll tackle a task with PowerShell. We’ll start with the high-level view of workflows and explain why you need them, their strengths and weaknesses, and the constraints you’ll face when using workflows.

**Note**

We’ll come back to this point a number of times throughout the chapter, but you need to be aware that although workflows look like PowerShell code, they’re not PowerShell—they’re code written with a PowerShell-like-syntax.

Before we get to the deep technicalities of PowerShell workflows, we need to give you the overview we promised.

## 12.1 Workflow Overview

In this section, we’ll give you an overview of when you should consider using workflows. We’ll then use the ever-popular “Hello world” approach to create your first workflow. The section closes by looking at the differences between workflows and PowerShell code, followed by the restrictions imposed on your code by using PowerShell.

Workflows introduce a number of keywords that we’ll explain. The foreach keyword you’ve seen already gains new functionality in workflows that we’ll demonstrate. Once you have a sound grasp of workflow features and syntax, we’ll provide some examples of using workflows. These will include nested workflows, how workflows interact with the PowerShell job engine, and the large set of parameters you can use with workflows. We’ll introduce and explain a number of cmdlets for working directly with workflows.

First, though, why should you think about using workflows?

### 12.1.1 Why Use Workflows

PowerShell workflows are designed for scenarios where you have processes that meet the following criteria:

* Run for a long time, potentially for days.
* Execute unattended.
* Run in parallel across one or more machines.
* Are interruptible. They can be stopped and restarted through the use of checkpoints, which means the state of the process can be persisted to disk.
* Need to survive a reboot of the system against which the workflow is executing.
* Need to track execution.

So, what kind of task would fit this model? A few real-world examples should help establish when using workflows would be beneficial:

* You need to test hundreds or thousands of servers for the presence of a particular piece of software—a typical compliance issue in many organizations.
* You need to make a series of modifications to the configuration of many servers that involves rebooting the servers in the middle of the process.
* You need to modify the configuration of multiple servers with a process that involves many steps. Capturing the state of the workflow with a series of checkpoints enables you to restart the process from the point of an error rather than from the beginning

By now you are probably thinking, “I can do that already.” That’s true; there are ways to solve all of these problems without using workflows, but workflows make it easier for you to solve these problems.

There’s nothing to stop you solving any of these problems by using non-workflow solutions, but we recommend that you definitely consider using workflows for these three areas:

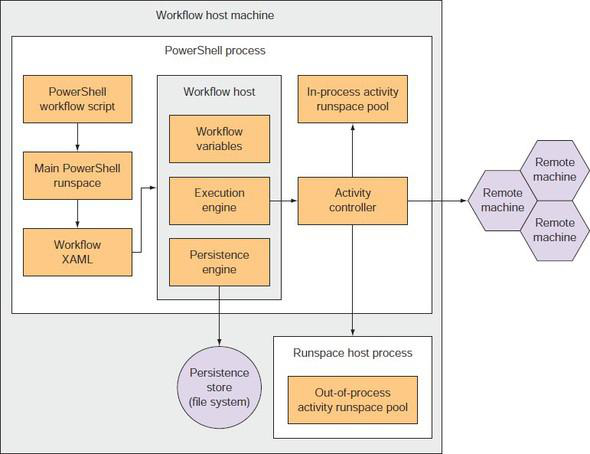
* You need to interrupt and restart tasks.
* You need to checkpoint the task (persistence).
* You have a mixture of sequential and parallel tasks.

You’ll see how workflows solve these problems as we progress through the chapter, but now it’s time to discover the architecture of PowerShell workflows.

### 12.1.2 Workflow Architecture

In this section, we’re going to look at the internal architecture of PowerShell workflow. Having some knowledge of the architecture will help you understand and predict the behavior of a workflow script. [Figure 12.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12fig01) shows all the pieces of the PowerShell workflow architecture.

Figure 12.1: PowerShell workflow architecture



This diagram is rather complex, so we’ll go through it one piece at a time. On the left side of the picture is a box labelled Main PowerShell runspace. This is your interactive PowerShell session. In workflows, it does only one thing: takes a PowerShell workflow script and translates it into workflow XAML (eXtensible Markup Language). The workflow engine requires this XAML representation in order to execute the workflow. The XAML is then passed to the workflow host component for execution. At this point, the involvement of the main runspace ends (other than to wait for the workflow to finish).

Inside the workflow host, the workflow execution engine is the element that executes the program logic, but there are a couple of other important parts. First is the variable store. The workflow engine has its own way of dealing with variables that’s completely separate from the PowerShell runspace variable store. (You’ll see why this matters when we get to writing workflows.) The other major component is the persistence store. One of the signature features of a workflow is that it can halt execution, save all state to the persistence, and be resumed at a later time. In the case of a PowerShell workflow, that persistence store is the file system. By default the persistence store is located at $env:LOCALAPPDATA\Microsoft\Windows\PowerShell\WF\PS. This location can be modified by the New-PSWorkflowExecutionOption cmdlet if required.

After receiving the XAML, the workflow execution engine starts processing the steps in the workflow. These individual steps are called activities, which include not only imperative actions but also the if statements, loops, and other flow-control statements that make up the workflow. PowerShell workflow exposes PowerShell commands as activities so the workflow engine can run them, but the control statements are specific to the workflow engine. For each PowerShell command, there is a corresponding activity wrapper that will call it. These wrappers are generated using tools that produce a C# wrapper for the command. The C# wrappers for all the commands in a module are compiled into a new binary module with the word activities inserted into the name (see [table 12.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12table02)).

To recap, the workflow engine runs the activity, which in turn runs the command. But as you know, PowerShell commands can run only in runspaces. This is shown on the right side of [figure 12.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12fig01). PowerShell workflow maintains several pools of runspaces that are used to execute the individual commands. Of particular interest is the out-of-process runspace pool. This is used by default and guarantees that a failing command won’t cause the workflow host process to fail. This approach makes the workflow execution significantly more reliable—but at a cost. Running commands out of process adds a lot of overhead because of the need to serialize data between the workflow host process and the out-of-process runspace host. To improve performance, you can also choose to run activities in process but at some risk of destabilizing the engine (usually a pretty small risk).

Also note that these are runspace pools, not individual runspaces, which is what allows for parallel operations. The workflow engine dispatches each command to a runspace in a pool and then waits until the operations are complete. If there are more operations to execute than there are available runspaces, then the workflow engine will queue the remaining commands and they’ll be processed as a runspace becomes available.

Okay, enough about architecture; let’s start writing workflows. We’ll refer to the architecture as needed in the remainder of the chapter.

### 12.1.3 Your First Workflow

In earlier chapters we introduced new functionality by creating a “Hello world” example. Workflows are no different. This is “Hello world” presented as a workflow:

workflow hello

{

'Hello World'

}

Executing this workflow gives the following output:

PS> hello

Hello World

If you’re thinking, “This looks like a function definition,” at this point you’re correct. In many cases, workflows look exactly like a function except that the function keyword is replaced by the workflow keyword. You can see if a function is a workflow by using Get-Command on the command name, which in this case gives the following:

PS> Get-Command hello

CommandType Name Version Source

----------- ---- ------- ------

Workflow hello

Let’s drill down into our hello workflow. A number of interesting features are exposed when you examine the full output from

PS> Get-Command hello | Format-List \*

#### Note

Running this command will give you all the information about the workflow, including the XAML representation of the workflow. The resulting output is too large to include in the book, but we’ve included a copy in the download available from the book’s website.

The XAML definition of the workflow is worth examining. If you look through the scriptblock definition you’ll see this line:

function hello {

When a PowerShell workflow is created, the command is persisted as a function. You can test this:

PS> Get-ChildItem -Path Function:\hello

CommandType Name Version Source

----------- ---- ------- ------

Workflow hello

The command type is set as a Workflow even though the command is in the function drive. Referring to the architecture, this function is the piece that runs in the main PowerShell runspace. Unlike a regular function, which executes the actions in its body, the job of a workflow function is to pass the XAML definition to the workflow engine, which ultimately does all the execution. This means that the function you defined is quite different than the function that’s run in the main runspace. You can view this generated function dispatcher function by running

PS> Get-Command hello | select -ExpandProperty Scriptblock

Once again, the output of this command is too long to include in the text of the book, but the bulk of the function is a large number of parameter definitions. You can look at these parameters by doing

PS> Get-Command hello | select -ExpandProperty Parameters

This list of parameters is shown in [table 12.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12table01).

|  |  |  |
| --- | --- | --- |
| **Table 12.1: Default workflow parameters** | | |
| PSParameterCollection | PSComputerName | PSCredential |
| PSConnectionRetryCount | PSConnectionRetry-IntervalSec | PSRunningTimeoutSec |
| PSElapsedTimeoutSec | PSPersist | PSAuthentication |
| PSAuthenticationLevel | PSApplicationName | PSPort |
| PSUseSSL | PSConfigurationName | PSConnectionURI |
| PSAllowRedirection | PSSessionOption | PSCertificate-Thumbprint |
| PSPrivateMetadata | AsJob | JobName |
| InputObject | ErrorAction | WarningAction |
| InformationAction | Verbose | Debug |
| ErrorVariable | WarningVariable | InformationVariable |
| OutVariable | OutBuffer | PipelineVariable |

You should recognize some of the parameters such as Verbose and Debug from Power-Shell functions. Others, such as PSPersist and PSPort, are new workflow-specific parameters that are automatically defined, and available, for every workflow you write without you having to do any extra work!

#### Note

The full definition of each of the workflow parameters can be found in the help file about\_WorkflowCommonParameters.

We’ve mentioned the XAML definition corresponding to the workflow function several times. This is the XamlDefinition that was generated for our hello workflow.

#### Listing 12.1: Xamldefinition of hello workflow

<Activity

x:Class="Microsoft.PowerShell.DynamicActivities.Activity\_1303329265"

xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"

xmlns:sad="clr-namespace:System.Activities.Debugger;assembly=

System.Activities"

xmlns:local="clr-namespace:Microsoft.PowerShell.DynamicActivities"

xmlns:mva="clr-namespace:Microsoft.VisualBasic.Activities;assembly=

System.Activities"

mva:VisualBasic.Settings="Assembly references and imported namespaces serialized as XML namespaces"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:ns0="clr-namespace:System;assembly=mscorlib"

xmlns:ns1="clr-namespace:Microsoft.PowerShell.Utility.Activities;

assembly=Microsoft.PowerShell.Utility.Activities"

xmlns:ns2="clr-namespace:Microsoft.PowerShell.Activities;

assembly=Microsoft.PowerShell.Activities"

xmlns:ns3="clr-namespace:System.Activities;assembly=System.Activities"

xmlns:ns4="clr-namespace:System.Management.Automation;

assembly=System.Management.Automation"

>

<Sequence>

<ns2:SetPSWorkflowData>

<ns2:SetPSWorkflowData.OtherVariableName>Position

</ns2:SetPSWorkflowData.OtherVariableName>

<ns2:SetPSWorkflowData.Value>

<ns3:InArgument x:TypeArguments="ns0:Object">

<ns2:PowerShellValue

x:TypeArguments="ns0:Object"

Expression="'2:2:hello'" />

</ns3:InArgument>

</ns2:SetPSWorkflowData.Value>

</ns2:SetPSWorkflowData>

<ns1:WriteOutput>

<ns1:WriteOutput.NoEnumerate>[

System.Management.Automation.SwitchParameter.Present]

</ns1:WriteOutput.NoEnumerate>

<ns1:WriteOutput.InputObject>

<InArgument x:TypeArguments="ns4:PSObject[]">

<ns2:PowerShellValue x:TypeArguments="ns4:PSObject[]"

Expression="'Hello World'" />

</InArgument>

</ns1:WriteOutput.InputObject>

</ns1:WriteOutput>

<Sequence.Variables>

<Variable Name="WorkflowCommandName"

x:TypeArguments="ns0:String" Default = "hello" />

</Sequence.Variables>

</Sequence>

</Activity>

The XAML consists of a series of definitions and then, starting at the <Sequence> tag, comes the body of the workflow. Reading through the Sequence section you’ll recognize a number of activities such as WriteOutput that correspond to PowerShell commands.

#### XAML, PowerShell, and workflows

XAML is the language used in Windows Workflow Foundation (WF), which is part of the .NET framework. Although XAML was primarily designed for creating GUIs, it’s also used as a common markup across multiple Microsoft products such as SharePoint and Team Foundation Server workflows.

The WF provides an API, a workflow engine, and a designer. Each step in your workflow is modeled as an activity—either from the .NET library or custom-created. Activities are assembled into workflows using the Workflow Designer in Visual Studio.

You can import PowerShell workflows into the Workflow Designer—see <http://mng.bz/473s>—but the result is not too intelligible because of all of the boilerplate code generated by PowerShell WF. One of the advantages that PowerShell brings to workflow is a much more concise notation for expressing workflows.

(Trivia: Windows Workflow Foundation is abbreviated as WF instead of WWF because WWF conflicted with the World Wildlife Federation trademark. Yes, wildlife, not wrestling.

Now that you’ve seen under the covers of a PowerShell workflow, how are the various parts generated?

### 12.1.4 Running a Workflow

When you run a PowerShell workflow, the built-in script-to-workflow compiler generates the XAML for you. The user experience is simplified by creating a PowerShell function (with the same parameters) that wraps the XAML. The function’s job is to coordinate the execution of the workflow within the PowerShell workflow engine.

A PowerShell workflow is executed as a PowerShell job (see [chapter 13](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-13-108.xhtml#ch13)), which provides the asynchronous capability of workflows. A job executes in a separate PowerShell process. You can test this by running the code in the following listing.

#### Listing 12.2: Demonstration of workflow PowerShell processes

workflow Invoke-ParallelForEach 1

{

foreach -parallel ($i in 1..10) 2

{

InlineScript 3

{

"foo: $using:i"

}

$count = Get-Process -Name PowerShell\* |

Measure-Object |

Select-Object -ExpandProperty Count

"Number of PowerShell processes = $count" 4

}

}

$startcount = Get-Process -Name PowerShell\* | 5

Measure-Object |

Select-Object -ExpandProperty Count

"Number of starting PowerShell processes = $startcount"

Invoke-ParallelForEach 6

* 1 Define workflow
* 2 Create loop
* 3 Output loop counter
* 4 Process count during execution
* 5 Initial process count
* 6 Execute workflow

#### Note

The workflow keywords in [listing 12.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12ex02) are explained in detail in [section 12.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-122-104.xhtml#ch12lev1sec2). For now, we’ll tell you what they do.

The workflow keyword 1 defines the start of the workflow. A foreach statement 2creates a loop. The –parallel parameter ensures the loop’s iterations are run in parallel rather than sequentially. An InlineScript 3 is used to write out the current iteration details and Get-Process 4 is used to determine the number of PowerShell processes in use.

Before the workflow is invoked Get-Process is used to determine the initial number of PowerShell processes 5. The workflow is invoked 6 and you’ll see something like the following:

Number of starting PowerShell processes = 2

foo: 9

foo: 6

foo: 10

foo: 7

foo: 3

Number of PowerShell processes = 7

Number of PowerShell processes = 7

Number of PowerShell processes = 7

foo: 4

foo: 5

foo: 2

Number of PowerShell processes = 7

Number of PowerShell processes = 7

Number of PowerShell processes = 7

foo: 8

foo: 1

Number of PowerShell processes = 7

Number of PowerShell processes = 7

Number of PowerShell processes = 7

Number of PowerShell processes = 7

The first thing to note is the output is not sequential. You have a mixture of the iteration number and the number of processes. Also note that the iteration numbers look random.

#### Note

This is extremely important to remember. When running tasks in parallel in a workflow, you have no control over the order in which data is returned. If you can’t identify where the data comes from, you won’t reap the benefit of running the workflow.

The second point to note is the number of PowerShell processes. It immediately jumps from 2 (at the start) to 7, suggesting that the workflow created another 5 processes and used them to perform its tasks. The PowerShell job system will automatically close the sessions that were created. This isn’t immediate; there’s a slight delay.

We’ve mentioned that you’re not using PowerShell cmdlets when writing workflows; you’re using workflow activities. What do we mean by that?

### 12.1.5 Cmdlets vs. Activities

When you look at [listing 12.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12ex02) you see a script written in PowerShell syntax. It uses some cmdlets including Get-Process, Measure-Object, and Select-Object. This all looks familiar, but as you know from your study of the workflow architecture, these aren’t cmdlets, and the important phrase in the first sentence of this section is “PowerShell syntax.” When you’re executing inside a workflow, you’re not using the PowerShell runtime (unless you’re in an InlineScript block) and you’re not using cmdlets directly; you’re using workflow activities, which are cmdlets with a WF wrapper, as you discovered in the architecture section.

The workflow activities are contained in assemblies (located in the global assembly cache) that correspond to PowerShell modules, as shown in [table 12.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12table02).

| **Table 12.2: PowerShell modules and corresponding activities** | |
| --- | --- |
| **PowerShell module** | **PowerShell activity assembly** |
| n/a | Microsoft.PowerShell.Activities |
| Microsoft.PowerShell.Core | Microsoft.PowerShell.Core.Activities |
| Microsoft.PowerShell.Diagnostics | Microsoft.PowerShell.Diagnostics.Activities |
| Microsoft.PowerShell.Management | Microsoft.PowerShell.Management.Activities |
| Microsoft.PowerShell.Security | Microsoft.PowerShell.Security.Activities |
| Microsoft.PowerShell.Utility | Microsoft.PowerShell.Utility.Activities |
| Microsoft.WSMan.Management | Microsoft.WSMan.Management.Activities |

The assembly Microsoft.PowerShell.Activities doesn’t have a corresponding cmdlet module because it contains a set of activities that are part of the PowerShell workflow runtime and is loaded when workflows are used.

Each activity has a similar (almost identical) syntax to the corresponding command because the activity wrappers are generated from the cmdlets directly; the properties on the generated activities are a strict superset of the parameters on the command. The difference is in the common parameters. [Table 12.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12table03) lists the common parameters found on workflow activities.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 12.3: Common activity parameters** | | | |
| AppendOutput | Debug | DisplayName | ErrorAction |
| Input | MergeError-ToOutput | PSActionRetry-Count | PSActionRetry-IntervalSec |
| PSActionRunning-TimeoutSec | PSApplication-Name | PSAuthentication | PSCertificate-Thumbprint |
| PSComputerName | PSConfiguration-Name | PSConnection-RetryCount | PSConnection-RetryIntervalSec |
| PSConnectionURI | PSCredential | PSDebug | PSDisable-Serialization |
| PSDisable-Serialization-Preference | PSError | PSPersist | PSPort |
| PSProgress | PSProgress-Message | PSRemoting-Behavior | PSRequired-Modules |
| PSSessionOption | PSUseSSL | PSVerbose | PSWarning |
| Result | UseDefaultInput | Verbose | WarningAction |

We’re not going to explain each parameter, but we do need to make a couple of points:

* PSActionRetryIntervalSec relates to retrying actions in the workflow, not network retries.
* PSRequiredModules is used by the activity wrapper generator code to tell the WF runtime what module it needs to load in order to run the wrapped command.

In many cases, the parameters match cmdlet parameters you’ve already seen. Notice that many of them have a PS prefix. This can cause confusion where a cmdlet has a –ComputerName parameter and the workflow activity has a –PSComputerName parameter. The parameters are described in detail in the about\_ActivityCommonParameters help file.

#### Note

You’ve been warned. You will trip over the difference in parameter names at some time when writing workflows.

One important point is that not all cmdlets have corresponding workflow activities. For example, this workflow will work:

workflow test1

{

Get-CimInstance -ClassName Win32\_ComputerSystem

}

test1

But see what happens if you try to use Format-Table:

workflow test1

{

Get-CimInstance -ClassName Win32\_ComputerSystem |

Format-Table Name, Model

}

test1

At line:3 char:5

+ Format-Table Name, Model

+ ~~~~~~~~~~~~~~~~~~~~~~~~

Cannot call the 'Format-Table' command. Other commands from this module have

been packaged as workflow activities, but this command was specifically

excluded. This is likely because the command requires an interactive

Windows PowerShell session, or has behavior not suited for workflows. To

run this command anyway, place it within an inline-script

(InlineScript { Format-Table }) where it will be invoked in isolation.

+ CategoryInfo : ParserError: (:) [], ParseException

+ FullyQualifiedErrorId : CommandActivityExcluded

The error message explains why you’re getting an error and how you can use the cmdlet in a workflow. [Table 12.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12table04) lists the unsupported cmdlet groups.

| **Table 12.4: Unsupported cmdlet groups** | |
| --- | --- |
| **Unsupported cmdlet (group)** | **Reason** |
| \*Alias, \*FormatData, \*History, \*Location, \*PSDrive, \*Transcript, \*TypeDate, \*Variable, Connect/Disconnect-Wsman | Change only PowerShell session, so not needed in workflow because each activity runs in its own runspace instance. |
| Show-Command, Show-ControlPanelItem, Get-Credential, Show-EventLog, Out-Gridview, Read-Host, Debug-Process | Workflows don’t support interactive cmdlets. |
| \*BreakPoint, Get-PSCallStack, Set-PSDebug | These commands are session-specific, but workflow commands each run in isolation. |
| \*Transaction | Workflows don’t support transactions. |
| Format\* | Workflows are intended to be run in a distributed and asynchronous manner. Excluding the formatting cmdlets ensures a remote scenario isn’t broken by accident. |
| \*PSsession, \*PSsessionoption | Remoting controlled by workflow. |
| Export-Console,Get-ControlPanelItem, Out-Default, Out-Null, Write-Host, Export-ModuleMember, Add-PSSnapin, Get-PSSnapin, Remove-PSSnapin, Trace-Command | These are excluded because they affect the current session, excluded because the workflow is non-interactive, excluded because the workflow handles remoting itself, or excluded because they might break the remote asynchronous pattern for workflow. |

Some cmdlets, by default, can only be executed locally in workflows, as listed in [table 12.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12table05).

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 12.5: Cmdlets that can only be executed locally in workflows** | | | |
| Add-Member | Compare-Object | ConvertFrom-Csv | ConvertFrom-Json |
| ConvertFrom-StringData | Convert-Path | ConvertTo-Csv | ConvertTo-Html |
| ConvertTo-Json | ConvertTo-Xml | ForEach-Object | Get-Host |
| Get-Member | Get-Random | Get-Unique | Group-Object |
| Measure-Command | Measure-Object | New-PSSessionOption | New-PSTransportOption |
| New-TimeSpan | Out-Default | Out-Host | Out-Null |
| Out-String | Select-Object | Sort-Object | Update-List |
| Where-Object | Write-Debug | Write-Error | Write-Host |
| Write-Output | Write-Progress | Write-Verbose |  |

If you need to use a cmdlet that doesn’t have a matching workflow activity or you want to execute one of the cmdlets from [table 12.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12table05) remotely, you have to use an InlineScript (see [section 12.2.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-122-104.xhtml#ch12lev2sec9)).

### 12.1.6 Workflow Restrictions

Given the architectural considerations of the PowerShell workflow runtime, there are a number of restrictions you need to be aware of in PowerShell workflows:

* Language restrictions
* Aliases and positional parameters
* Object serialization and deserialization
* Variable usage

We’ll be covering variable usage in [section 12.3.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-123-105.xhtml#ch12lev2sec11).

#### Language Restrictions

A number of PowerShell keywords and techniques aren’t supported in workflows. [Table 12.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12table06) provides a summary.

|  |  |  |
| --- | --- | --- |
| **Table 12.6: PowerShell language and techniques not supported in workflows** | | |
| Begin, Process, End | Break, Continue | Subexpressions |
| Multiple assignment | Modify loop variable | Dynamic parameters |
| Set properties | Dot-sourcing | Advanced parameter validation |
| Single #requires | Switch statement | Trap statement |
| Inline help | Setting drive qualified variables | Method invocation on objects |
| New-Object with –COMobject parameter |  |  |

Some of the language restrictions can be overcome by using an InlineScript block, as you’ll see later, but the others you need to avoid. They’re a restriction imposed by Windows Workflow Foundation, and you’ll get an error if you try to use them.

#### Using Aliases in Workflows

When workflows were first introduced in PowerShell v3 this would have failed:

workflow test1

{

gps powershell\*

}

test1

The reason for the failure was twofold:

* Aliases weren’t allowed.
* Positional parameters weren’t allowed.

In addition, in PowerShell v3 you had to use full parameter names. These restrictions were relaxed in PowerShell v4. PowerShell workflows aren’t a command-line activity, and its best practice in your scripts to not use aliases, positional parameters, or parameter abbreviations, so we recommend that you adhere to the original restriction.

The objects you receive from a workflow add an additional restriction.

#### Objects Returned from Workflows

Workflows use PowerShell remoting for access to remote machines, so the objects returned to you have been serialized and then deserialized. A deserialized object gives you the properties of the object but not the methods—it’s inert. Lots of PowerShell code does something like this:

$prc = Get-Process -Name notepad

$prc.Kill()

You create an object and then call a method on that object. This approach isn’t going to work in your workflows:

workflow test2

{

$prc = Get-Process -Name notepad

$prc.Kill()

}

It will throw an error about method invocation not being supported:

At line:4 char:3

+ $prc.Kill()

+ ~~~~~~~~~~~

Method invocation is not supported in a Windows PowerShell Workflow.

To use .NET scripting, place your commands in an inline script:

InlineScript { <commands> }.

+ CategoryInfo : ParserError: (:) [], ParentContainsErrorRecordException

+ FullyQualifiedErrorId : MethodInvocationNotSupported

The InlineScript activity is your get-out-of-jail card for a lot of workflow issues—you’ll see how it works later and how to perform this technique.

We’ve spent some time explaining how workflows look like PowerShell, but are different, yet use a PowerShell-like syntax. It’s time to dig deeper into workflow syntax and see how to use the workflow keywords.

## 12.2 Workflow Keywords

You need to understand a number of keywords in order to get the most out of workflows. These keywords are valid only inside the body of a workflow function. They enable parallel or sequential execution of commands, enable execution of cmdlets that don’t have workflow activities, allow embedding pure PowerShell code in workflows, and allow parallel execution inside a foreach loop. Each of these keywords will have a block containing one or more commands.

One of the major reasons for using a workflow is to enable commands to execute in parallel, so we’ll start with that.

### 12.2.1 Parallel

By default, the commands within a workflow execute in sequence. Run the following workflow:

workflow p1

{

foreach ($i in 1..4){$i}

foreach ($j in 4..1){$j}

}

p1

You’ll see the screen output count up from 1 to 4 and then down from 4 to 1. That’s exactly the same output as if you’d run the commands in a function:

function f1 {

foreach ($i in 1..4){$i}

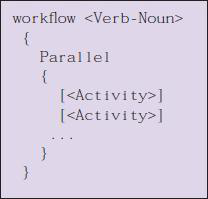
foreach ($j in 4..1){$j}

}

f1

If you need simultaneous execution of commands, you have to use the parallel keyword to instruct the workflow to run commands in parallel. The syntax is shown in [figure 12.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-122-104.xhtml#ch12fig02).

Figure 12.2: Syntax of the parallel keyword



Using the parallel keyword, our previous example becomes this:

workflow p2 {

parallel {

foreach ($i in 1..4){$i}

foreach ($j in 4..1){$j}

}

}

"$(p2)"

This time you’ll see a mixture of numbers counting up and counting down, something like this:

1 4 2 3 3 2 4 1

Parallel execution can be more efficient and can reduce run times:

PS> Measure-Command {p1} | Select Milliseconds

PS> Measure-Command {p2} | Select Milliseconds

Milliseconds

------------

269

160

You can see that the workflow executing commands in parallel finished in a shorter time. This is governed by the hardware where the workflow is running. Also, parallelism is much more effective in the remoting scenario where the real work is done on the remote machine. This is the primary scenario for using it. Simple local parallelism in a workflow isn’t terribly useful because there’s a lot of overhead in running a workflow.

The ability to execute commands in parallel is great, but as we showed earlier, you have no control over the order in which commands are executed and so you can’t predict the order in which the results will be returned. Sometimes you need to be able to control the order in which commands execute.

### 12.2.2 Sequence

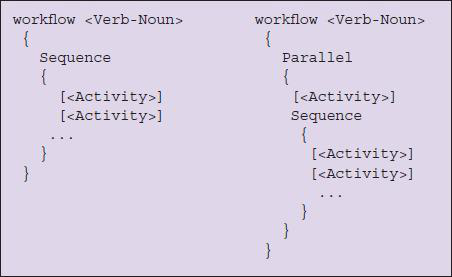
The sequence keyword is used to run a set of activities in sequence—in the order in which they are written, which is the default for workflow, so the sequence statement is intended to be nested inside parallel blocks so that you can execute multiple statements in parallel. An activity in a sequence scriptblock will execute only when the preceding activity has completed.

#### Note

When you include a sequence block in a parallel block, you have no control over when the sequence block executes within the parallel block. You can control the execution only within the sequence block.

[Figure 12.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-122-104.xhtml#ch12fig03) shows alternate syntaxes for using the sequence block.

Figure 12.3: Alternate syntaxes of a workflow sequence block



This needs an example to clarify. Consider this workflow:

workflow ps1

{

parallel {

foreach ($i in 1..4){$i}

sequence {

foreach ($k in 65..68){[char][byte]$k}

foreach ($k in 87..90){[char][byte]$k}

}

foreach ($j in 4..1){$j}

}

}

"$(ps1)"

Executing this workflow will produce the following output:

1 A 4 2 B 3 3 C 2 4 D 1 W X Y Z

The workflow from [section 12.2.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-122-104.xhtml#ch12lev2sec7) has been modified by inserting a sequence block between the two foreach blocks. The workflow will execute the foreach blocks and the sequence block in parallel. The commands inside the sequence block execute in order.

The outputs of the foreach blocks and the sequence block are intermixed, as you’d expect from parallel execution. The important point is that the output from the sequence block is ordered as expected, namely A–D followed by W–Z.

You now know how to execute commands in parallel and force execution to be sequential when required. In the next section, you’ll learn about dealing with cmdlets and PowerShell features that aren’t supported by workflows.

### 12.2.3 InlineScript

Workflows are written in PowerShell that has new keywords and some language restrictions. How do you incorporate “pure, traditional” PowerShell into a workflow?

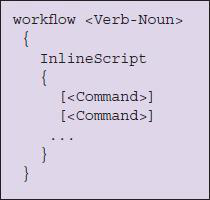
This is where InlineScript comes to the rescue. An InlineScript block can contain any and all valid PowerShell commands irrespective of their being normally supported in workflows.

#### Note

In many cases InlineScript (or inline functions) are the only practical way to use workflow. Using the workflow activity to get a registry key is ludicrously slow. Workflow is best used to sequence largish blocks of code that you don’t want to repeat.

You can use an InlineScript block in the main body of the workflow, inside a loop or control statement, or nested inside a parallel or sequential block. The syntax is illustrated in [figure 12.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-122-104.xhtml#ch12fig04).

Figure 12.4: Workflow InlineScript syntax



An InlineScript block has the activity common parameters including –PSPersist, but the PowerShell commands inside the InlineScript block don’t gain any of the activity common parameters or workflow features such as checkpointing.

Variables defined in a workflow aren’t visible to an InlineScript block, but the $using scope modifier can be used to access those variables; see [section 12.3.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-123-105.xhtml#ch12lev2sec12).

Using an InlineScript block is illustrated here.

#### Listing 12.3: Using an InlineScript block

workflow is1

{

parallel {

'BootTime from Parallel:' 1

Get-CimInstance -ClassName Win32\_OperatingSystem `

-PSComputerName $env:COMPUTERNAME |

Select-Object -ExpandProperty LastBootUpTime

InlineScript { 2

$os = Get-WmiObject -Class Win32\_OperatingSystem `

-ComputerName $env:COMPUTERNAME

'BootTime from InlineScript: '

$($os.ConvertToDateTime($os.LastBootUpTime))

}

}

}

is1

* 1 Parallel block
* 2 InlineScript block

Running the workflow gives these results:

BootTime from Parallel:

16 April 2017 22:45:29

BootTime from InlineScript:

16 April 2017 22:45:29

The parallel block 1 uses Get-CimInstance to retrieve the Win32\_OperatingSystem WMI class and return the LastBootUpTime property. The property is returned as a date as shown in the output (one of the reasons for using the CIM cmdlets rather than the WMI cmdlets). We’re using a workflow activity in this block (the –PSComputerName parameter).

#### Note

When creating workflows, the PowerShell ISE IntelliSense will work out if you’re using a workflow activity or a PowerShell cmdlet and show you the correct parameters.

Conversely, in the InlineScript block 2 we’re using a PowerShell cmdlet, Get -WmiObject. It also retrieves the Win32\_OperatingSystem WMI class but has to use the ConvertToDateTime method to return the date in a readable format. We deliberately wrapped the method in a subexpression to show PowerShell functionality normally not supported in workflows. We’re also using the –ComputerName parameter on Get -WmiObject as you’d expect.

You’ve seen workflows that execute in parallel and sequentially, but we haven’t looked at iterating over collections yet. Guess what’s next!

### 12.2.4 Foreach -parallel

Loops are an important part of coding; they enable you to repeatedly execute a set of commands with minimal coding effort. We discussed the standard PowerShell looping constructs in [chapter 5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-5-42.xhtml#ch05). Those constructs can also be used in PowerShell workflows.

#### Listing 12.4: Using loops in workflows

workflow fe

{

'Do loop'

$i = 1

$j = @()

do {

$j += $i

$i++

} while ($i -le 10)

"$j"

'While loop'

$i = 1

$j = @()

while ($i -le 10) {

$j += $i

$i++

}

"$j"

'For loop'

$j = @()

for ($i = 1; $i -le 10; $i++) {

$j += $i

}

"$j"

'Foreach loop'

$j = @()

foreach ($i in 1..10){$j += $i}

"$j"

}

fe

When you execute the workflow, you’ll see results like this:

Do loop

1 2 3 4 5 6 7 8 9 10

While loop

1 2 3 4 5 6 7 8 9 10

For loop

1 2 3 4 5 6 7 8 9 10

Foreach loop

1 2 3 4 5 6 7 8 9 10

You’ll immediately notice that each individual loop executes sequentially. You could try to put the loop inside a parallel block, but it wouldn’t make any difference.

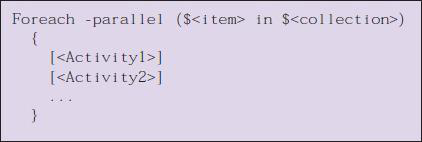
In the examples in [listing 12.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-122-104.xhtml#ch12ex04) we’re only listing numbers, so parallel versus sequential processing isn’t a great issue. If, on the other hand, you’re iterating through a collection of computers and needing to perform some actions on them, such as setting a registry key or pulling WMI data, being able to process the loop in parallel would be a significant time saver.

#### Note

If all the tasks in your workflow need to access the same set of remote machines, use the –PSComputerName parameter on the workflow, which will force parallel processing across the machines for each command.

The answer is to use foreach with the –parallel parameter, as shown in [figure 12.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-122-104.xhtml#ch12fig05).

Figure 12.5: Foreach –parallel syntax



As an example consider the following:

workflow fep {

foreach -parallel ($i in 1..10){$i}

}

"$(fep)"

This gave the following results when we tested it:

10 9 8 7 6 5 4 3 1 2

You can see that the results aren’t sequential when compared to the results obtained from [listing 12.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-122-104.xhtml#ch12ex04). The –parallel parameter runs the commands in the script block once, in sequence, for each item in the collection; the parallelization occurs at the item level. The collection must be created, and the variable defined, before the foreach –parallel statement.

This example is more practical:

workflow fs {

$fileshares = Get-FileShare

foreach -parallel ($fileshare in $fileshares){

InlineScript {

Get-Volume -FileShare $using:fileshare |

Select-Object @{N='Share'; E={$using:fileshare.Name}},

DriveLetter, FileSystem, HealthStatus,

@{N='FreePercent';

E={[math]::Round(($($\_.SizeRemaining) / $($\_.Size)) \* 100, 2)}}

}

}

}

fs

The collection of file shares on a machine is generated using Get-FileShare. For each file share in the collection, the volume data is retrieved and displayed. The shares are processed in parallel. Note that because the InlineScript activity runs code in a separate runspace (see [figure 12.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12fig01)), as in remoting, the $using: scope modifier is required to access the $fileshare variable. Each share produces results similar to this:

Share : C$

DriveLetter : C

FileSystem : NTFS

HealthStatus : Healthy

FreePercent : 39.51

PSComputerName : localhost

PSSourceJobInstanceId : ac7a4655-397c-483c-be73-6db80e4ae204

Notice that the computer name and the job instance identifier are automatically added.

You now have a good understanding of how workflows are constructed and how they work. Let’s look at how you use them.

## 12.3 Using Workflows Effectively

In this section, we’ll look at the parameters available on workflows; this is a separate but overlapping set of parameters to those available on individual activities. We’ll then look at using variables in workflows and the scoping issues this introduces, followed by showing you how workflows can be called from other workflows and even nested. We’ve said that workflows are run as jobs and we’ll cover that in [chapter 13](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-13-108.xhtml#ch13).

First, you need to know about the parameters available on workflows.

### 12.3.1 Workflow Parameters

Workflows have a large number of parameters by default. You saw the list of parameters in [section 12.1.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12lev2sec2). The default parameters can be split into two sets, as shown by the syntax of a simple workflow:

PS> workflow test {'Hello'}

PS> Get-Command test -Syntax

test [<WorkflowCommonParameters>] [<CommonParameters>]

The common parameters are those that you also see on functions and cmdlets. These are listed in [table 12.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-123-105.xhtml#ch12table07).

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 12.7: Workflow common parameters** | | | |
| InputObject | ErrorAction | WarningAction | InformationAction |
| Verbose | Debug | ErrorVariable | WarningVariable |
| InformationVariable | OutVariable | OutBuffer | PipelineVariable |

You’ve seen these parameters in use throughout the previous chapters. Much more interesting are the parameters that are unique to workflows, as presented in [table 12.8](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-123-105.xhtml#ch12table08).

#### Note

Workflows that are nested three or more levels deep don’t support any common parameters.

|  |  |  |
| --- | --- | --- |
| **Table 12.8: Parameters unique to workflows** | | |
| PSParameterCollection | PSComputerName | PSCredential |
| PSConnectionRetryCount | PSConnectionRetry-IntervalSec | PSRunningTimeoutSec |
| PSElapsedTimeoutSec | PSPersist | PSAuthentication |
| PSAuthenticationLevel | PSApplicationName | PSPort |
| PSUseSSL | PSConfigurationName | PSConnectionURI |
| PSAllowRedirection | PSSessionOption | PSCertificateThumbprint |
| PSPrivateMetadata | AsJob | JobName |

These parameters are described in the help file about\_WorkflowCommonParameters. You should compare the contents of [tables 12.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-123-105.xhtml#ch12table07) and [12.8](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-123-105.xhtml#ch12table08) with [table 12.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12table03) (activity common parameters) to see the differences and overlaps.

#### Note

Workflow (and activity) common parameters are all optional and named. None of them can be used as a positional parameter. They also don’t take input from the pipeline.

We won’t describe all these parameters in detail because many overlap with the remoting session parameters you’ve already seen—which is not surprising because workflows use WS-MAN to communicate with remote machines. Some of the parameters need to be discussed, starting with the way you pass computer names to workflows.

#### PSComputerName

This parameter specifies a list of computers on which the workflow will be run. You can use the name, IP address, or fully qualified domain name with the same approach as PowerShell remoting; that is, if you use an IP address you have to supply the appropriate credentials and the remote computer must use HTTPS or the IP address must be in the trusted hosts list.

All workflows and activities have -PSComputerName available as a parameter, so where should you put it? As usual, it depends.

If you put it at the workflow level

workflow test-remoteaccess {

Get-WmiObject -Class Win32\_ComputerSystem

}

test-remoteaccess -PSComputerName W16TGT01, W16DSC01

you’ll receive results like these for each machine:

Domain : Manticore.org

Manufacturer : Microsoft Corporation

Model : Virtual Machine

Name : W16TGT01

PrimaryOwnerName : Windows User

TotalPhysicalMemory : 1116749824

PSComputerName : localhost

Notice that the Get-WmiObject activity has no mention of remote machines. This is one advantage of using –PSComputerName at the workflow level in that you can easily use the same workflow locally and add the –PSComputerName parameter when you need to access remote machines.

Compare this to running Get-WmiObject directly against the local machine:

PS> Get-WmiObject -Class Win32\_ComputerSystem

Domain : Manticore.org

Manufacturer : Microsoft Corporation

Model : Virtual Machine

Name : W16AS01

PrimaryOwnerName : Windows User

TotalPhysicalMemory : 2429566976

The workflow adds a PSComputerName property to the output. This is the name of the computer on which you’re running the workflow. It is not the name of the remote machine even though the workflow has a –PSComputerName parameter! The remote machine name is in the Name property.

#### Note

This is one of those confusing points you’ll have to remember.

Using –PSComputerName at the workflow level is probably best kept for situations where you have simple data return requirements or you’re predominantly performing actions against the remote machine with minimal or no data returned.

When you use the –PSComputerName parameter, it effectively replaces the –ComputerName parameter on the cmdlet. You don’t get free connectivity! You’ll also find that you’re connecting over the native mechanism used by the cmdlet that corresponds to the workflow activity. If the remote machine doesn’t support that particular mode of connectivity, your workflow will fail for that machine.

Moving the –PSComputerName to the activity results in this code:

workflow test-remoteaccess {

param(

[string[]]$computername

)

foreach -parallel ($computer in $computername) {

Get-WmiObject -Class Win32\_ComputerSystem -PSComputerName $computer

}

}

test-remoteaccess -computername W16TGT01, W16DSC01

This workflow defines a parameter that takes a list of computer names. The foreach –parallel statement is used to iterate over the computer names. The computers in the list are processed in parallel, and the commands within the foreach –parallel block are processed sequentially for each computer. You’re back to using the native connectivity (DCOM in this case). This approach would be useful when you have a number of activities in your workflow, not all of which need to access a remote machine.

You need to consider one last scenario: running a workflow with an InlineScript block where the cmdlets in the block need to connect to remote machines. The big thing for you to remember in this scenario is that you’re running cmdlets, not workflow activities, so you need to use the cmdlet’s native parameter –ComputerName.

workflow Test-RemoteAccess

{

param(

[string[]]$computername

)

inlinescript {

foreach ($computer in $using:computername) {

Get-WmiObject -Class Win32\_ComputerSystem -ComputerName $computer

}

}

}

test-remoteaccess -computername W16TGT01, W16DSC01

The workflow has a -computername parameter that takes a list of computer names. Within the InlineScript block a foreach loop iterates over the list of computers. You have to define the foreach loop like this:

foreach ($computer in $using:computername)

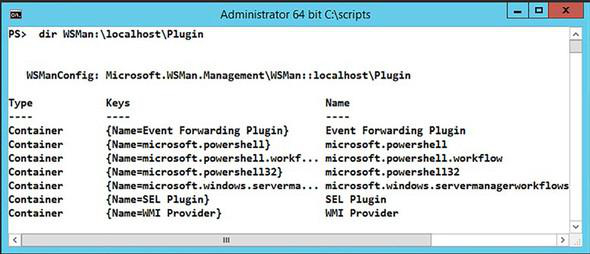
The $using modifier enables the loop to access the variable that was defined in a higher scope within the workflow; you’ll learn about that in the next section.

You will have to decide, based on what your workflow is doing, how you will pass computer names into the workflow and which parameters you need to use.

#### PSConfigurationName

This parameter specifies the session configuration used when connecting to remote computers. The default is Microsoft.PowerShell.Workflow, as shown in [figure 12.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-123-105.xhtml#ch12fig06).

Figure 12.6: Remoting endpoints on a Windows Server 2012 R2 system



The endpoints shown in [figure 12.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-123-105.xhtml#ch12fig06) are created by default when PowerShell remoting is enabled either explicitly by running Enable-PSRemoting or implicitly through installation of Windows Server 2012 (or later). Microsoft.PowerShell.Workflow is used by workflows, whereas Microsoft.PowerShell and Microsoft.PowerShell32 are used by PowerShell remoting.

You can drill down further into the endpoint configuration by examining the contents of the InitializationParameters, Resources, and Quotas containers.

The vast majority of the time you can use the default endpoint. If you need to create a new endpoint, you can use the approach outlined in [section 11.6.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11lev2sec28).

Possibly the most interesting parameter is –Persist, but we’ll delay talking about that until [chapter 13](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-13-108.xhtml#ch13) because we first need to look at using variables in workflows.

### 12.3.2 Variables in Workflows

You’ve seen variables used in various workflows in the earlier sections of this chapter. We’ve also shown you the $using and $workflow scoping modifiers that are used in workflows. Now it’s time to bring this together and show the rules for using variables in workflows. It’s not complicated—honest!

Workflows use lexical scoping, so a variable defined inside a block is visible only in that block and nested blocks. This leads to the following restrictions on the use of variables:

* Variables defined in a higher scope are visible to lower workflow scopes but not InlineScript scopes.
* You can’t have a variable in a lower scope with the same name as a variable in a higher scope—an aspect of lexical scoping caused by the underlying workflow engine.
* If you define or redefine a variable, you can use it in that scope without problems.
* There is no $global scope because workflows always run as jobs, so they have a new global context.
* Use the $workflow scope modifier to access, or modify, a variable defined in a higher scope, except in InlineScript blocks.
* Use the $using scope modifier in InlineScript blocks to access, or modify, variables defined in a higher scope.
* Modification of a variable from a higher scope in an InlineScript requires the use of a temporary variable.
* You can’t use subexpressions in workflows.

That probably sounds confusing, so let’s work through an example of using variables in workflows that’ll bring the rules into focus.

#### Listing 12.5: Using variables in workflows

workflow demo-scope

{

# This is a workflow top-level variable

$a = 22

"Initial value of A is: $a"

# Access $a from Inlinescript (bringing a workflow

#variable to the PowerShell session) using $using

inlinescript {"PowerShell variable A is: $a"}

inlinescript {"Workflow variable A is: $using:a"}

## changing a variable value

$a = InlineScript {$b = $Using:a+5; $b}

"Workflow variable A after InlineScript change is: $a"

parallel {

sequence {

# Reading a top-level variable

"Value of A inside parallel is: $a"

# Updating a top-level variable with

# $workflow:<variable name>

$workflow:a = 3

}

}

"Updated value of A is: $a"

}

demo-scope

When you run this workflow, you should see this output:

Initial value of A is: 22

PowerShell variable A is:

Workflow variable A is: 22

Workflow variable A after InlineScript change is: 27

Value of A inside parallel is: 27

Updated value of A is: 3

The workflow starts by defining a variable - $a = 22 and then displaying its value. In an InlineScript if you try to access a variable defined in a higher scope, you get nothing, as shown in the second line of the output. You have to use $using:a to access the variable. If you want to change that variable, you’ll have to use a second variable and return it to the original variable:

$a = InlineScript {$b = $Using:a+5; $b}

The output shows the variable now has a value of 27.

Moving into the parallel block, you can read the variable without any scope issues. If you need to change the variable’s value, you can access it via the $workflow scope modifier. The bottom line with variables in workflows is keep it simple and be careful.

So far, you’ve seen single workflows. When using scripts or functions, you can call other scripts or functions (or even nest functions). How do workflows handle this?

### 12.3.3 Nested Workflows

Think about how you use your PowerShell scripts and functions; you probably build a number of functions that you reuse and call from other functions and scripts. The whole concept of reusability should permeate your PowerShell code so that you maximize the return from the time and effort you put into developing your code. How can PowerShell functions and PowerShell workflows be used inside other workflows?

The mechanisms available to reuse existing functionality break down into three broad groups:

* PowerShell workflows called from your workflow or nested in your workflow
* PowerShell functions either in the same script file as the workflow or through a PowerShell module
* PowerShell scripts on the local or remote machine

Let’s start by looking at how your workflow can interact with other workflows using a practical example from standard Active Directory administration tasks. It’s generally regarded as good practice to clean up the accounts in your Active Directory. You would normally look at disabled accounts, expired accounts, and accounts with passwords that never expire. You can make a decision on what to do with each account once you’ve identified accounts that match your criteria.

To find disabled accounts, run the following:

PS> Search-ADAccount -AccountDisabled |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\DisabledAccounts.csv -NoTypeInformation

To find expired accounts, do this:

PS> Search-ADAccount -AccountExpired |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\ExpiredAccounts.csv -NoTypeInformation

To find accounts whose passwords never expire, use this:

PS> Search-ADAccount -PasswordNeverExpires |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\PsswdNeverExpireAccounts.csv `

-NoTypeInformation

These three simple scripts will be familiar to Active Directory administrators. Using these is more efficient that trying to perform the task by hand, but you have to run them sequentially. Can workflows help you introduce some parallelism?

The most direct approach would be to wrap the scripts into a single workflow:

workflow Get-ADReport

{

parallel {

Search-ADAccount -AccountDisabled |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\DisabledAccounts.csv `

-NoTypeInformation

Search-ADAccount -AccountExpired |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\ExpiredAccounts.csv `

-NoTypeInformation

Search-ADAccount -PasswordNeverExpires |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\PsswdNeverExpireAccounts.csv `

-NoTypeInformation

}

}

Get-ADReport

The three CSV files are produced more or less simultaneously. This is great if this task runs once in a while but not if you need a more granular approach and want to run each search individually. One approach that also has the benefit of making maintenance easier is to move the individual workflows out of the main workflow, like this:

workflow get-disabled

{

Search-ADAccount -AccountDisabled |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\DisabledAccounts.csv `

-NoTypeInformation

}

workflow get-expired

{

Search-ADAccount -AccountExpired |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\ExpiredAccounts.csv `

-NoTypeInformation

}

workflow get-passwordneverexpire

{

Search-ADAccount -PasswordNeverExpires |

Select-Object -Property DistinguishedName |

Export-Csv -Path c:\ADReports\PsswdNeverExpireAccounts.csv `

-NoTypeInformation

}

workflow Get-ADReport

{

parallel {

get-disabled

get-expired

get-passwordneverexpire

}

}

Get-ADReport

You can take this a stage further and separate your workflows into individual files and create a .psm1 file to load them as a module. You can then add further functionality in a granular manner without affecting the bulk of your code.

#### Note

In reality, nested workflows can cause problems if you try to nest complicated workflows. The preferred and recommended reuse strategy is to use functions instead of nested workflows.

Functions can be embedded in a workflow or in the script used to create the workflow. In the following workflow, a list of computer names is passed in through the -computerName parameter. A foreach –parallel loop is used to iterate over the list of computers. Test-Connection is used to determine if the remote system is contactable, and if so, the function is called.

In this case, the function is defined outside the workflow. You could as easily have defined it inside the workflow. Similarly, you could put the functions into a separate script and load them and the workflow as part of a module.

The important point is that the workflows or functions you want to call are loaded, or defined, before you want to use them:

function get-fcomputersystem {

param ([string]$fcomputer)

Get-WmiObject -Class Win32\_ComputerSystem -ComputerName $fcomputer

}

workflow get-computersystem

{

param([string[]]$computerName)

## Alternate location for function

# The contents of the foreach block will be executed in parallel

foreach -parallel($computer in $computerName) {

if (Test-Connection -ComputerName $computer -Quiet -Count 1) {

get-fcomputersystem -fcomputer $computer

}

else {

"$computer unreachable"

}

}

}

Get-ComputerSystem -ComputerName $ENV:COMPUTERNAME

Scripts are the third and last of the methods you can utilize to reuse existing code. Take the three scripts utilizing Search-ADAccount introduced at the top of the section and put each into a script file:

get-disabledaccount.ps1

get-expiredaccount.ps1

get-passwordNexpire.ps1

You still want these to run in parallel, so you might try this:

workflow get-ADReport

{

parallel {

c:\adreports\get-disabledaccount.ps1

c:\adreports\get-expiredaccount.ps1

c:\adreports\get-passwordNexpire.ps1

}

}

Unfortunately, this won’t work and you’ll see an error:

At line:3 char:4

+ c:\adreports\get-disabledaccount.ps1

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Cannot find the 'c:\adreports\get-disabledaccount.ps1' command. If this

command is defined as a workflow, ensure it is defined before the workflow

that calls it. If it is a command intended to run directly within

Windows PowerShell (or is not available on this system), place it in an

InlineScript: 'InlineScript { c:\adreports\get-disabledaccount.ps1 }'

+ CategoryInfo : ParserError: (:) [], ParseException

+ FullyQualifiedErrorId : CommandNotFound

You want the scripts to run using an InlineScript block but also ensure parallelism, so run each script separately:

workflow get-ADReport {

parallel {

inlinescript {c:\adreports\get-disabledaccount.ps1}

inlinescript {c:\adreports\get-expiredaccount.ps1}

inlinescript {c:\adreports\get-passwordNexpire.ps1 }

}

}

What about the situation where you want to run a script that exists on a remote system? The answer is to put the scripts in the C:\ADReports folder on the remote machine and run your local workflow as shown here:

Get-ADReport –PSComputerName W16TGT01

The scripts will run on the remote machine and, because you haven’t modified them, that’s where the output will be produced.

#### Workflows as jobs

We’ve said that workflows use the PowerShell job engine a number of times and that workflows are interruptible. We’ll look at those two ideas in [chapter 13](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-13-108.xhtml#ch13).

You now have a good understanding of workflows and how to use them. The workflows you’ve seen so far have been scripts. A few cmdlets are also available for working with workflows.

## 12.4 Workflow Cmdlets

PowerShell provides you with two workflow modules:

PS> Get-Module -ListAvailable \*workflow\* | Format-Table -AutoSize

Directory: C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules

ModuleType Version Name ExportedCommands

---------- ------- ---- ----------------

Manifest 2.0.0.0 PSWorkflow {New-PSWorkflowExecutionOption,

New-PSWorkflowSession, nwsn}

Manifest 1.0.0.0 PSWorkflowUtility Invoke-AsWorkflow

New-PSWorkflowExecutionOption and New-PSWorkflowSession from the PSWorkflow module are analogous to the remoting management cmdlets New-PSSessionOption and New-PSSession respectively. Invoke-AsWorkflow is a way to test your code in a workflow without any further modification.

Let’s start by looking at the options you have for executing workflows.

### 12.4.1 Workflow Execution Options

The syntax for New-PSWorkflowExecutionOption is shown in [figure 12.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-124-106.xhtml#ch12fig07).

Figure 12.7: Syntax of the New-PSWorkflowExecutionOption cmdlet



An object created with New-PSWorkflowExecutionOption is used to configure the options for workflow sessions. You’ll learn how to create a session for workflows in the next section. The cmdlet common parameters are available on this cmdlet but not the workflow common parameters. [Table 12.9](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-124-106.xhtml#ch12table09) lists the parameters and their meaning.

| **Table 12.9: New-PSWorkflowExecutionOption parameters** | |
| --- | --- |
| **Parameter** | **Meaning** |
| PersistencePath | Path on disk for storing persistence data. Default is $env:LocalAppData\Microsoft\Windows\PowerShell\WF\PS. Persistence data is created when a workflow is checkpointed or suspended (see [chapter 13](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-13-108.xhtml#ch13)). |
| MaxPersistenceStoreSizeGB | Maximum space, in GB, allocated to workflows running in the session. Default is 10 GB. If the size is exceeded, the store is expanded and warnings are displayed. |
| PersistWithEncryption | Encrypts data in persistence store. Default is false. |
| MaxRunningWorkflows | Maximum number of running workflows in session. Default is 30. |
| AllowedActivity | Namespace qualified activities that can be run in the session. Wildcards are allowed. Default is built-in WF activities and activities matching PowerShell core cmdlets. |
| OutOfProcessActivity | Which allowed activities (specified in AllowedActivity) are run out of process. Default is InlineScript. |
| EnableValidation | Verifies all workflow activities in session are included in allowed activities list. Default is true. |
| MaxDisconnectedSessions | Maximum number of remote sessions that are in disconnected state across all remote computers. Default is 100. |
| MaxConnectedSessions | Maximum number of remote sessions that are in operational state across all remote computers. Default is 100. |
| MaxSessionsPerWorkflow | Maximum number of sessions created to support each workflow. Default is 5. |
| MaxSessionsPerRemoteNode | Maximum number of sessions that can be connected to each remote computer. Default is 5. |
| MaxActivityProcesses | Maximum processes that can be created in a session to support workflow activities. Default is 5. |
| ActivityProcessIdleTimeoutSec | Determines the time before an activity host process is closed once the process becomes idle. Default is 60 seconds. |
| RemoteNodeSessionIdleTimeoutSec | Specifies timeout on an idle session connected to a remote computer. Default is 60 seconds. |
| SessionThrottleLimit | Number of operations created to support all workflows started in a session. Default is 100. |
| WorkflowShutdownTimeoutMSec | Time session is maintained after all workflows are forcibly suspended. Default is 500 seconds. |

As an example, we’ll modify the code from [listing 12.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-121-103.xhtml#ch12ex02) to give this:

workflow Invoke-ParallelForEach

{

foreach -parallel ($i in 1..10)

{

InlineScript

{

"foo: $using:i"

}

$count = Get-Process -Name PowerShell\* |

Measure-Object |

Select-Object -ExpandProperty Count

"Number of PowerShell processes = $count"

}

}

You can create a new workflow execution option object like this:

PS> $wfopt = New-PSWorkflowExecutionOption -MaxSessionsPerWorkflow 20 `

-MaxSessionsPerRemoteNode 20 -MaxActivityProcesses 20

PS> $wfopt

SessionThrottleLimit : 100

PersistencePath : C:\Users\Richard\AppData\Local\

Microsoft\Windows\PowerShell\WF\PS

MaxPersistenceStoreSizeGB : 10

PersistWithEncryption : False

MaxRunningWorkflows : 30

AllowedActivity : {PSDefaultActivities}

OutOfProcessActivity : {InlineScript}

EnableValidation : True

MaxDisconnectedSessions : 1000

MaxConnectedSessions : 100

MaxSessionsPerWorkflow : 20

MaxSessionsPerRemoteNode : 20

MaxActivityProcesses : 20

ActivityProcessIdleTimeoutSec : 60

RemoteNodeSessionIdleTimeoutSec : 60

WorkflowShutdownTimeoutMSec : 500

You can then create a new endpoint:

PS> Register-PSSessionConfiguration -Name PiAWorkflows `

-SessionTypeOption $wfopt -SessionType Workflow -Force

WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin

Type Keys Name

---- ---- ----

Container {Name=PiAWorkflows} PiAWorkflows

The workflow endpoints can be explicitly accessed using the –PSConfiguration parameter on your workflow. The default endpoint can be accessed by leaving the parameter off or explicitly providing the endpoint name:

PS> Invoke-ParallelForEach `

-PSConfigurationName Microsoft.PowerShell.Workflow

If you want to access your new endpoint, then use its name:

PS> Invoke-ParallelForEach -PSConfigurationName PiAWorkflows

You can modify an endpoint:

PS> Set-PSSessionConfiguration -Name PiAWorkFlows `

-SessionTypeOption (New-PSWorkflowExecutionOption `

-SessionThrottleLimit 500) -Force

You can review the change:

PS> Get-PSSessionConfiguration -Name PiAWorkflows |

select SessionThrottleLimit

SessionThrottleLimit

--------------------

500

If you need to remove a workflow endpoint, use this command:

PS> UnRegister-PSSessionConfiguration -Name PiAWorkflows –Force

#### Note

You can modify the settings on the default workflow endpoint, but they work well in all but the most exceptional cases. If you want to experiment, it’s best to create a new endpoint.

You’ve seen how to use workflows against remote machines using the –PSComputerName parameter. You can also create workflow sessions to remote computers in a similar manner to PowerShell remoting.

### 12.4.2 Workflow Sessions

Workflow sessions are similar to the remoting sessions you saw in [chapter 11](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-11-93.xhtml#ch11). You use them to create a permanent connection to the remote machine rather than creating and destroying connections as needed.

Creating a session is similar to PowerShell remoting:

PS> $wfs = New-PSWorkflowSession -ComputerName W16TGT01

As with PowerShell remoting, the commands you call have to exist on the remote machine or you must supply them to the session. You’ll create a scriptblock containing your workflow (including the command to execute the workflow):

$sb = {

workflow Invoke-ParallelForEach

{

foreach -parallel ($i in 1..10)

{

InlineScript

{

"foo: $using:i"

}

$count = Get-Process -Name PowerShell\* |

Measure-Object |

Select-Object -ExpandProperty Count

"Number of PowerShell processes = $count"

}

}

Invoke-ParallelForEach

}

The scriptblock is executed through the session:

PS> Invoke-Command -Session $wfs -ScriptBlock $sb

When the workflow commences, you’ll see a warning like this:

WARNING: [localhost]:This workflow job cannot be suspended because there are

no checkpoints (also called persistence points) in the workflow. To make

the workflow job suspendable, add checkpoints to the workflow.

For more information about how to add checkpoints, see the help topics for

Windows PowerShell Workflow.

When you’ve finished with your session, its best practice to remove it:

PS> Remove-PSSession -Session $wfs

#### Using non-default workflow endpoints remotely

New-PSWorkflowSession doesn’t give you a way to access any workflow endpoints other than the default one. The –SessionOption takes remoting options, not workflow execution options, from New-PSWorkflowExecutionOption!

If you need to access a workflow endpoint that you’ve created, you need to use New-PSsession to create the session. In this case we’ve used the technique from the previous section to create an endpoint called PiAWorkflows on a remote machine. A remote session is created to the endpoint:

PS>$ts = New-PSSession -ComputerName W16TGT01 `

-ConfigurationName PiAWorkflows

PS>Invoke-Command -Session $ts -ScriptBlock $sb

When you run the workflow, you’ll see that the number of processes being used has increased. In our test we saw this:

Number of PowerShell processes = 11

Workflow sessions provide another option when running against remote machines. You can run workflows through standard remoting sessions, in which case they’ll use the default workflow endpoint on the remote machine.

PowerShell, and therefore workflow, remoting works against the machine or machines to which you’ve connected, but if you try to connect to a third machine from your remote machine, you’ll hit the double-hop problem.

#### Double-Hop Problem

The usual scenario in remote administration is that you’re working locally on machine A and connect remotely to machine B to perform one or more tasks. If you try to perform an action on machine C from your session on machine B, you’ll receive an error. Let’s see what happens. Start with a standard call to a remote machine:

PS> Invoke-Command -ScriptBlock {Get-Process lsass} `

-ComputerName W16TGT01

This works as expected. Now try accessing another machine from the session on W16TGT01:

PS> Invoke-Command -ScriptBlock {

Invoke-Command -ScriptBlock {

Get-Process lsass } -ComputerName W16DSC01

} -ComputerName W16TGT01

[W16DSC01] Connecting to remote server W16DSC01 failed with the following

error message : WinRM cannot process the request. The following error

with errorcode 0x8009030e occurred while using Kerberos authentication:

A specified logon session does not exist. It may already have been

terminated. Possible causes are:

-The user name or password specified are invalid.

-Kerberos is used when no authentication method and no user name are specified.

-Kerberos accepts domain user names, but not local user names.

-The Service Principal Name (SPN) for the remote computer name and port does not exist.

-The client and remote computers are in different domains and there is no trust between the two domains.

After checking for the above issues, try the following:

-Check the Event Viewer for events related to authentication.

-Change the authentication method; add the destination computer to the WinRM TrustedHosts configuration setting or use HTTPS transport.

Note that computers in the TrustedHosts list might not be authenticated.

-For more information about WinRM configuration, run the following command: winrm help config. For more

information, see the about\_Remote\_Troubleshooting Help topic.

+ CategoryInfo : OpenError: (W16DSC01:String) [], PSRemotingTransportException

+ FullyQualifiedErrorId : 1312,PSSessionStateBroken

+ PSComputerName : W16TGT01

This rather long error message boils down to saying that Kerberos authentication failed to connect you to the second machine: your credentials weren’t available to the session on machine B (W16TGT01) when it attempted to create a session on machine C (W16DSC01).

One solution to this problem is to use the Credential Security Support Provider (CredSSP), but because that involves sending your password in clear text across the network, this solution isn’t acceptable to many organizations. A more acceptable solution is use the RunAS configuration option on a PowerShell remoting session. First, create a credential object on machine B for the account you’ll use to connect to machine C:

PS> $cred = Get-Credential manticore\richard

Then use that credential when you create the remoting endpoint on machine B:

PS> Register-PSSessionConfiguration -Name DHsol -RunAsCredential $cred

WARNING: When RunAs is enabled in a Windows PowerShell session configuration,

the Windows security model cannot enforce a security boundary between

different user sessions that are created by using this endpoint. Verify

that the Windows PowerShell runspace configuration is restricted to only

the necessary set of cmdlets and capabilities.

WARNING: Register-PSSessionConfiguration may need to restart the WinRM

service if a configuration using this name has recently been unregistered,

certain system data structures may still be cached. In that case, a

restart of WinRM may be required.

All WinRM sessions connected to Windows PowerShell session configurations,

such as Microsoft.PowerShell and session configurations that are created

with the Register-PSSessionConfiguration cmdlet, are disconnected.

Reading the warning that’s issued when you create an endpoint with a credential gives you some additional information.

#### Note

The credential used for the endpoint is stored as an encrypted secure string on the machine.

You can now use the endpoint and successfully perform a double hop:

PS> $tsd = New-PSSession -ComputerName W16TGT01 -ConfigurationName DHsol

PS> Invoke-Command -ScriptBlock {Invoke-Command -ScriptBlock {Get-Process

lsass } -ComputerName W16DSC01 } -Session $tsd

Remote access for workflows works the same way. Use the -RunAsCredential parameter when you create a new workflow endpoint on machine B:

PS> Register-PSSessionConfiguration -Name PiAWorkflows `

-RunAsCredential $cred -SessionType Workflow -Force

Alternatively, if you have an existing endpoint, you can modify it to add a credential:

PS> Set-PSSessionConfiguration -Name PiAWorkflows -RunAsCredential $cred

Modify your workflow to access a remote machine by adding the -PSComputerName parameter to the Get-Process activity:

$sb = {

workflow Invoke-ParallelForEach

{

foreach -parallel ($i in 1..10)

{

InlineScript

{

"foo: $using:i"

}

$count = Get-Process -Name PowerShell\* -PSComputerName W16DSC01 |

Measure-Object |

Select-Object -ExpandProperty Count

"Number of PowerShell processes = $count"

}

}

Invoke-ParallelForEach

}

Re-create the remote session and run your workflow:

PS> $ts = New-PSSession -ComputerName W16TGT01 `

-ConfigurationName PiAWorkflows

PS> Invoke-Command -Session $ts -ScriptBlock $sb

Your workflow will now perform the double hop and connect to the third machine. The drawback to this technique is that you need to maintain the credential used on the endpoint. If the password changes, you need to update the endpoint with the new credential. If you need to configure a number of machines in this manner, consider using a service account approach and use an account with a strong password that’s changed infrequently.

### 12.4.3 Invoking as Workflow

Creating and testing workflows is a nontrivial task, but help is available within PowerShell through the Invoke-AsWorkflow cmdlet in the PSWorkflowUtility module.

The cmdlet runs any command or expression as an inline script in a workflow. You get these benefits of workflows: interruptability, persisting, tracking, and the workflow common parameters. You don’t get access to the parallel or foreach –parallel options.

These examples show how to use the cmdlet:

PS> Invoke-AsWorkflow -CommandName Get-Process `

-Parameter @{Name = 'powershell'}

PS> Invoke-AsWorkflow -Expression 'ping 127.0.0.1'

This concludes our examination of PowerShell workflows, a powerful tool that in the correct circumstances can be an efficient way to work with multiple remote machines.

## 12.5 Summary

* PowerShell v3 introduced the workflow keyword, and although workflows are written with a PowerShell-like syntax, they aren’t PowerShell.
* Workflows are excellent when you need to interrupt tasks or have mixture of parallel and sequential tasks.
* Workflows execute as PowerShell jobs.
* You have no control over the order in which data is returned when running tasks in parallel in a workflow.
* Workflows use workflow activities that correspond to PowerShell cmdlets, but not all cmdlets have corresponding activities because they may not make sense in the context of a workflow.
* Workflows and workflow activities have overlapping sets of common parameters.
* A number of PowerShell language options and techniques aren’t supported in workflows.
* Workflows run tasks sequentially by default.
* Use the parallel block to run commands in parallel.
* Use the sequence block to run commands sequentially inside a parallel block.
* An InlineScript block can run standard PowerShell commands, including those not supported in workflows.
* Foreach -parallel iterates over a collection of objects in parallel.
* The workflow engine allows for nested lexical scope, which PowerShell does not support by default, so the $workflow: scope modifier is required to modify a variable defined in a higher scope.
* When using the InlineScript activity, you need to use the $using: prefix. Note that $using: variables are read-only. Workflow-scope variables can’t be modified in an InlineScript activity.
* You can’t use subexpressions in workflows.
* Workflows can be nested and called from other workflows, but it’s not recommended that workflows be nested to more than two levels because of the overhead of workflow calling workflow.
* New-PSWorkflowExecutionOption can create an object to configure workflow endpoints.
* A new workflow endpoint can be created with Register-PSSessionConfiguration.
* -PSConfigurationName allows you to specify the workflow endpoint to use.
* Workflow sessions are remoting sessions. Use Set-PSSessionConfiguration to modify a workflow endpoint.
* New-PSWorkflowSession connects to the default workflow endpoint. New-PSSession is used to connect to non-default endpoints, including any end user–created workflow endpoints.
* Invoke-AsWorkflow is used to run PowerShell commands and expressions as workflows, providing all the benefits of workflow execution without having to create a workflow to wrap a single command.

In the next chapter, we’ll build on what you saw in this chapter when we look at pausing workflows and dig further into PowerShell jobs.

## Chapter 13: PowerShell Jobs

### Overview

This chapter covers

* Asynchronous processing
* PowerShell jobs
* Job cmdlets
* Job types
* Workflow jobs
* Scheduled jobs

*Exit pursued by a bear  
—William Shakespeare Stage directions from*The Winter’s Tale

So far, most of the techniques we’ve shown you have been synchronous, meaning you type in the command—be it a cmdlet, script, or function—and wait for the results. The results are back in a few seconds at most, usually far quicker. Synchronous execution is perfect for ad hoc, interactive working, but what about when you’re executing a long-running process against many remote machines that could take hours to run? Waiting for those to finish locks you out of further work in that console. You could open additional instances of PowerShell or you could run the tasks asynchronously.

An asynchronous task is one that’s started and left to run to completion in the background as you carry on working at other tasks. Asynchronous execution is supplied to PowerShell by using PowerShell jobs.

**Note**

In PowerShell v2, PowerShell jobs depended heavily on PowerShell remoting. Those dependencies were removed in PowerShell v3.

In this chapter, we’ll show you how to use PowerShell’s job engine to perform tasks asynchronously. We’ll start with a look at the types of jobs available in PowerShell and the cmdlets you can use to work with jobs. We’ll build on [chapter 12](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-12-102.xhtml#ch12) and show how PowerShell jobs are used to manage interruptions to workflows including reboots. We’ll close the chapter by looking at how you can combine PowerShell jobs with the scheduler to perform asynchronous tasks without manual intervention—great for those long jobs that run through the middle of the night.

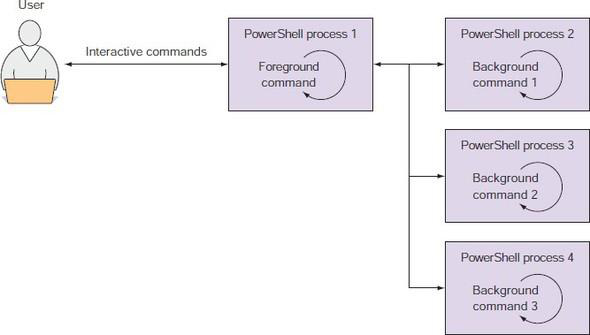
## 13.1 Background Jobs in PowerShell

When you run a command in a PowerShell session, the session is effectively blocked until the command completes and returns its results (or fails). You’re prevented from running new commands until the command completes. If you change things so that the caller doesn’t block, then other commands can run in parallel. This is how PowerShell background jobs work. With background jobs, the arrangement of executing commands and processes is shown in [figure 13.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-131-109.xhtml#ch13fig01).

#### Note

Some commands have built-in job support. For example, the WMI commands have an -AsJob parameter that allows one or more WMI operations to execute in the background. This type of job doesn’t rely on the background-execution mechanism we’re describing in this section. Instead, it uses its own implementation of background execution. In the case of WMI jobs, they run in a new process but on a separate thread. The PowerShell job infrastructure was explicitly designed to support this kind of extension. If third parties expose their job abstractions as subclasses of the PowerShell Job type, these extension jobs can be managed using the built-in job cmdlets like native PowerShell jobs.

Figure 13.1: The user sends interactive commands to be executed by the foreground loop. Background commands are executed in separate processes; each process has its own command loop. For each background job the user creates, a new instance of PowerShell.exe is run to host the command loop for that job. This means that if there are three background jobs as shown, then four processes are running—three for the background jobs and one for the interactive foreground job



There’s more to background jobs than executing multiple things at the same time. Background jobs are designed to be commands that run asynchronously while you continue to do other things at the console. This means there needs to be a way to manage these background jobs—starting and stopping them as well as retrieving the output in a controlled way.

#### Note

Background jobs are implemented using processes that are children of your interactive PowerShell process. This means that if you end your PowerShell session, causing the process to exit, this will also cause all the background jobs to be terminated, because child processes are terminated when the parent process exits.

In this section, we’ll cover the cmdlets that are used to manage background jobs. We’ll look at starting, stopping, and waiting for jobs. We’ll explore the Job objects used to represent a running job. Finally, you’ll learn how to combine remoting with jobs to run jobs on remote machines.

### 13.1.1 The Job Commands

PowerShell jobs are managed with a set of cmdlets, shown in [table 13.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-131-109.xhtml#ch13table01).

| **Table 13.1: The cmdlets for working with PowerShell jobs** | |
| --- | --- |
| **Cmdlet** | **Description** |
| Start-Job | Used to start background jobs. It takes a scriptblock as the argument representing the job to execute. |
| Stop-Job | Stops a job based on the JobID. |
| Get-Job | Returns a list of currently executing jobs associated with the current session. |
| Wait-Job | Waits for one or more jobs to complete. |
| Receive-Job | Gets the results for a specific job. |
| Remove-Job | Removes a job from the job table so the resources can be released. |
| Debug-Job | Debugs the scriptblock executed by a job. See section 15.6.1. |
| Suspend-Job | Saves the state and pauses execution of a job—only used with workflow jobs. See [section 13.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-132-110.xhtml#ch13lev1sec2). |
| Resume-Job | Restarts a previously suspended job—only used with workflow jobs. See [section 13.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-132-110.xhtml#ch13lev1sec2). |

A background job runs commands asynchronously. It’s used to execute long-running commands in a way that the interactive session isn’t blocked until that command completes.

#### Note

You can use the PowerShell APIs explained in [chapter 20](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-20-150.xhtml#ch20) as another way to perform actions asynchronously.

When a synchronous command runs, PowerShell waits until that command has completed before accepting any new commands. When a command is run in the background, instead of blocking, the command returns immediately, emitting an object that represents the new background job.

Although you get back control immediately (a new prompt) with the Job object, you obviously won’t get the results of that job even if the job runs quickly. Instead, you use a separate command to get the job’s results. You also have commands to stop the job, to wait for the job to be completed, and to delete the job. Let’s see how these commands are used.

### 13.1.2 Working with the Job Cmdlets

You use the Start-Job command to start a background job on a local computer. Let’s try this with a simple example that prints a string to the console.

#### Note

We’re deliberately using simple examples so that the job concepts are stressed rather than the clever code the job is running. In the real world, you wouldn’t run most of our examples as jobs.

You’ll start a job and then pipe the resulting Job object through Format-List so you can see all of the members on the object:

PS> Start-Job -ScriptBlock {'Hi'} | Format-List

HasMoreData : True

StatusMessage :

Location : localhost

Command : 'Hi'

JobStateInfo : Running

Finished : System.Threading.ManualResetEvent

InstanceId : 7590daa0-de23-4b65-ae4c-6c69970399f1

Id : 3

Name : Job3

ChildJobs : {Job4}

PSBeginTime : 09/05/2017 10:31:15

PSEndTime :

PSJobTypeName : BackgroundJob

Output : {}

Error : {}

Progress : {}

Verbose : {}

Debug : {}

Warning : {}

Information : {}

State : Running

As with the remoting cmdlets, the command to execute is specified by a scriptblock. When the command runs, you see that an object is returned, containing a wealth of information about the job. We’ll look at this object in detail later on. For now, we’ll keep looking at the cmdlets. Now that you’ve started a job, you can use the Get-Job cmdlet to get information about that job:

PS> Get-Job | Format-List

HasMoreData : True

StatusMessage :

Location : localhost

Command : 'Hi'

JobStateInfo : Completed

Finished : System.Threading.ManualResetEvent

InstanceId : 7590daa0-de23-4b65-ae4c-6c69970399f1

Id : 3

Name : Job3

ChildJobs : {Job4}

PSBeginTime : 09/05/2017 10:31:15

PSEndTime : 09/05/2017 10:31:15

PSJobTypeName : BackgroundJob

Output : {}

Error : {}

Progress : {}

Verbose : {}

Debug : {}

Warning : {}

Information : {}

State : Completed

This cmdlet returned the same Job object that you saw returned from Start-Job. (You can tell it’s the same object by looking at the InstanceId, which is a GUID and is guaranteed to be unique for each job.) There’s one significant difference in this output: If you look at the State property, you’ll see that it has changed from Running to Completed.

The first thing to note is that a job remains in the job table even after it has completed and will remain there until it’s explicitly removed using the Remove-Job cmdlet. To get the results of the job, you can use another cmdlet: Receive-Job. This cmdlet will return the results of the command that was executed:

PS> Receive-Job -Id 3

Hi

This returns the string that was emitted by the scriptblock passed to Start-Job. Using Receive-Job in this manner strips the data from the completed job. If you want to retain the data on the job for future processing, you need to use the -Keep parameter:

PS> Start-Job -ScriptBlock {'Hi'}

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

5 Job5 BackgroundJob Running True localhost 'Hi'

PS> Receive-Job -Id 5 -Keep

Hi

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

3 Job3 BackgroundJob Completed False localhost 'Hi'

5 Job5 BackgroundJob Completed True localhost 'Hi'

Notice the HasMoreData property is False for job Id 3 and True (you used the -Keep parameter) for job Id 5.

#### Note

Depending on the activity on your system the jobs you run may have a different Id and Name to those we show.

This isn’t an interesting example, so let’s try something that will take a bit longer to run. First, define the scriptblock you want to run in the $jsb variable:

PS> $jsb = {

foreach ($i in 1..10) { Start-Sleep 5; "i is $i" }

}

Now start the job. The command is too big to display in the default formatting. The compressed output doesn’t matter because the only thing you want at this point is the job’s Id:

PS> Start-Job -ScriptBlock $jsb

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

9 Job9 BackgroundJob Running True localhost ...

Wait 10 seconds or so and start calling Receive-Job with the job’s Id:

PS> Receive-Job 9

i is 1

i is 2

The first call returned the first 2 items out of the 10 you’re expecting. Wait another 10 seconds and call it again

PS> Receive-Job 9

i is 3

i is 4

i is 5

and you get another three items. Keep calling it until you get all the items:

PS> Receive-Job 9

i is 6

i is 7

PS> Receive-Job 9

i is 8

i is 9

i is 10

PS> Receive-Job 9

PS>

This last call didn’t return anything because the job has completed and all items have already been returned. You can verify this by calling Get-Job

PS> Get-Job 9

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

9 Job9 BackgroundJob Completed False localhost ...

and you see that its state is Completed. Because the job is running asynchronously, the number of items that are returned depends on when you call Receive-Job.

#### Waiting for Jobs to Complete

So how do you wait until the job has completed? You could write a loop to keep checking the State property, but that would be annoying and inefficient. Instead, you can use the Wait-Job cmdlet:

PS> $jb = Start-Job $jsb; Wait-Job $jb ; Receive-Job $jb

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

11 Job11 BackgroundJob Completed True localhost ...

i is 1

i is 2

i is 3

i is 4

i is 5

i is 6

i is 7

i is 8

i is 9

i is 10

In this example, you’re capturing the job object emitted by Start-Job in the $jb variable so you can use it in the subsequent Wait-Job and Receive-Job commands. Because of the Wait-Job, when you call Receive-Job you get all the input.

#### Note

If you use Wait-Job, the PowerShell session is blocked until the job is completed.

Notice that Wait-Job returns the object representing the job that has finished. You can use this to simplify the example a bit:

PS> Start-Job $jsb | Wait-Job | Receive-Job

i is 1

i is 2

i is 3

i is 4

i is 5

i is 6

i is 7

i is 8

i is 9

i is 10

In this example, Start-Job passes the Job object to Wait-Job. When the job completes, Wait-Job passes the Job object to Receive-Job to get the results. This eliminates the need for an intermediate variable.

#### Removing Jobs

So far, you’ve been creating jobs but haven’t removed any. This means that when you call Get-Job, you’ll see that there are a number of jobs still in the job table:

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

3 Job3 BackgroundJob Completed False localhost 'Hi'

5 Job5 BackgroundJob Completed True localhost 'Hi'

7 Job7 BackgroundJob Completed False localhost ...

9 Job9 BackgroundJob Completed False localhost ...

11 Job11 BackgroundJob Completed False localhost ...

13 Job13 BackgroundJob Completed False localhost ...

Each time you start a job, it gets added to the job table. You can clean things up using the Remove-Job cmdlet. To empty the table, use Remove-Job with a wildcard:

PS> Remove-Job \*

or you could use this:

PS> Get-Job | Remove-Job

Now when you call Get-Job, nothing is returned. This is probably not the best way to clean things up. A better solution would be to look for jobs that have completed and have no more data. That would look like the following:

function Clear-CompletedJobs {

Get-Job |

where { $\_.State -eq "Completed" -and -not $\_.HasMoreData } |

Remove-Job

}

This function calls Get-Job to get the list of all jobs, filters that list based on the State and HasMoreData properties, and then pipes the filtered list into Remove-Job. By doing this, only completed jobs for which all data has been received will be removed. This allows you to clean up the job table without worrying about losing information or getting errors. If you do want to kill all the jobs immediately, you can use the -Force parameter on Remove-Job.

Running the Clear-CompletedJobs function will remove all the jobs in the previous job table except job Id 5. You can remove an individual job:

PS> Remove-Job -Id 5

PS> Get-Job

PS>

In the next section, we’ll look at ways you can apply concurrent jobs to solve problems.

### 13.1.3 Working with Multiple Jobs

So far, we’ve looked at simple patterns working with one job at a time, but you can run a number of jobs at the same time. Doing so complicates things—you have to be able to handle the output from multiple jobs, but you get the benefit of running tasks in parallel, which makes you more efficient. Let’s look at how to do this.

#### Listing 13.1: Example of running multiple jobs

1..5 | foreach {

Start-Job -name "job$\_" -ScriptBlock {

param($number)

$waitTime = Get-Random -Minimum 4 -Maximum 10

Start-Sleep -Seconds $waitTime

"Job $number is complete; waited $waitTime"

} -ArgumentList $\_ > $null }

Wait-Job job\* | Receive-Job

This example starts a number of jobs that will run concurrently, waits for all of them to complete, and then gets all the results. Run the code in [listing 13.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-131-109.xhtml#ch13ex01) and you’ll see results like this:

Job 1 is complete; waited 9

Job 2 is complete; waited 5

Job 3 is complete; waited 8

Job 4 is complete; waited 9

Job 5 is complete; waited 7

As you can see, all the results are captured, ordered by the job name. Now let’s look at a more useful application of this pattern. This listing shows a function that searches multiple directories in parallel looking for a specific pattern.

#### Listing 13.2: A function that searches a collection of folders in parallel

function Search-FilesInParallel

{

param (

[parameter(mandatory=$true, position=0)]

$Pattern,

[parameter(mandatory=$true, position=1)]

[string[]]

$Path,

[parameter()]

$Filter = "\*.txt",

[parameter()]

[switch]

$Any

)

$jobid = [Guid]::NewGuid().ToString() 1

$jobs = foreach ($element in $path)

{

Start-Job -name "$Srch{jobid}" -scriptblock { 2

param($pattern, $path, $filter, $any)

Get-ChildItem -Path $path -Recurse -Filter $filter

Select-String -list:$any $pattern 3

} -ArgumentList $pattern,$element,$filter,$any

}

Wait-Job -any:$any $jobs | Receive-Job 4

Remove-Job -force $jobs

}

* 1 Generate GUID to use for job ID.
* 2 Start search job for each path
* 3 Pass -any switch to Select-String.
* 4 Wait for any or all jobs.

This function takes a list of folder paths to search, along with a pattern to search for. By default, the function will only search TXT files. It also has a switch, -Any, that controls how the search is performed. If the switch isn’t specified, all matches from all folders will be returned. If it’s specified, only the first match will be returned and the remaining incomplete jobs will be canceled.

This function seems like a useful tool. Unfortunately, jobs are implemented by creating new processes for each job, and this is an expensive operation—so expensive, in fact, that generally it’s much slower than searching all the files serially.

#### Note

If the creation of the new process is a significant fraction of your job run time, then you probably don’t need to use a job. Keep jobs for long-running tasks.

In practice, PowerShell jobs are a way of dealing with latency (the time it takes for an operation to return a result) and not throughput (the amount of data that gets processed). This is a good trade-off for remote management tasks when you’re talking to many machines more or less at once. The amount of data is frequently not large, and the overall execution time is dominated by the time it takes to connect to a remote machine. With that in mind, let’s look at how remoting and jobs work together.

### 13.1.4 Starting Jobs on Remote Computers

PowerShell is designed for administering remote computers, so it follows that you can also create and manage jobs on remote computers.

#### Note

To work with remote jobs, remoting must be enabled on the remote machine.

The easiest way to do this is to use the -AsJob parameter on Invoke-Command. Alternatively, the scriptblock passed to Invoke-Command can call Start-Job explicitly. Let’s see how this works.

#### Child Jobs and Nesting

So far we’ve talked about Job objects as atomic—one Job object per job. In practice, it’s a bit more sophisticated than that. There are scenarios when you need to be able to aggregate collections of jobs under a single master, or executive, job. We’ll get to those situations soon. For now, know that background jobs always consist of a parent job and one or more child jobs.

For jobs started using Start-Job or the -AsJob parameter on Invoke-Command, the parent job is the executive. It doesn’t run any commands or return any results.

#### Note

The executive does no work—it supervises. All the work is done by the subordinates. That sounds familiar somehow ….

This collection of child jobs is stored in the ChildJobs property of the parent Job object. The child Job objects have a name, ID, and instance ID that differ from the parent job so that you can manage the parent and each child job individually or as a single unit.

To see the parent and all the children in a Job, use the Get-Job cmdlet to get the parent Job object and then pipe it to Format-List, which displays the Name and ChildJobs as properties of the objects. Here’s what that looks like:

PS> Get-Job | Format-List -Property Name, ChildJobs

Name : Job3

ChildJobs : {Job4}

You can also use a Get-Job command on the child job, as shown in the following command

PS> Get-Job -Name Job4

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

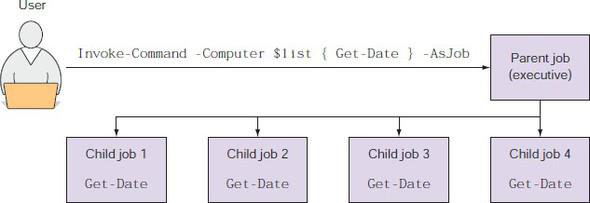
4 Job4 Completed True localhost 'Hi'

and so on until you get to a Job that has no children.

#### Child Jobs with Invoke-Command

Let’s look at the scenario where you need to have more than one child job. When Start-Job is used to start a job on a local computer, the job always consists of the executive parent job and a single child job that runs the command. When you use the -AsJob parameter on Invoke-Command to start a job on multiple computers, you have the situation where the job consists of an executive parent job and one child job for each command running on a remote server, as shown in [figure 13.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-131-109.xhtml#ch13fig02).

Figure 13.2: The relationship between the executive job and the nested jobs created when Invoke-Command -AsJob is used to run commands on multiple remote computers. The user calls Invoke-Command to start a job with multiple nested jobs, one for each target node in $list



When you use Invoke-Command to explicitly run Start-Job on the remote machines, the result is the same as a local command run on each remote computer. The command returns a Job object for each computer. The Job object consists of an executive parent job and one child job that runs the command.

The parent job represents all the child jobs. When you manage a parent job, you also manage the associated child jobs. For example, if you stop a parent job, all child jobs are also stopped. Similarly, when you get the results of a parent job, you’re also getting the results of all child jobs.

Most of the time, you don’t need to be concerned with the fact that there are parent and child jobs, but it’s possible to manage the child jobs individually. This approach is typically used only when you want to investigate a problem with a job or get the results of only one of a number of child jobs started by using the -AsJob parameter of Invoke-Command.

The following command uses Invoke-Command with -AsJob to start background jobs on the local computer and two remote computers. The command saves the job in the $j variable:

PS> $j = Invoke-Command -ComputerName localhost, W16DC01, W16TGT01 `

-ScriptBlock {Get-Date} -AsJob

When you display the Name and ChildJob properties of the object in $j, it shows that the command returned a Job object with three child jobs, one for each computer:

PS> $j | Format-List Name, ChildJobs

Name : Job1

ChildJobs : {Job2, Job3, Job4}

When you display the parent job, it shows that the overall job was considered to have failed:

PS> $j

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

1 Job1 RemoteJob Failed True localhost,W16DC01... Get-Date

But on further investigation, when you run Get-Job on each of the child jobs, you find that only one of them has failed:

PS> Get-Job -Name job2, job3, job4

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

2 Job2 Failed False localhost Get-Date

3 Job3 Completed True W16DC01 Get-Date

4 Job4 Completed True W16TGT01 Get-Date

#### Note

The job running on localhost failed because the PowerShell console wasn’t running with elevated privileges. Use Run As Administrator to start PowerShell when you want to access the local machine in this manner.

To get the results of all child jobs, use the Receive-Job cmdlet to obtain the results of the parent job. But you can also get the results of a particular child job, as shown in the following command:

PS> Receive-Job -Id 3 -Keep |

Format-Table PSComputerName, DateTime -AutoSize

PSComputerName DateTime

-------------- --------

W16DC01 09 May 2017 10:50:19

In this example, you’re using the -Keep parameter, which allows you to read, but not remove, output from a job. When you use -Keep, the output from the job is retained in the output buffer for that job. You’re using it here so that when you do a Receive-Job on the executive job, you’ll get the output of all jobs in a single collection. In effect, this is a way of peeking at the output of one of the child jobs. By using child jobs, you have much more granular control over the set of activities you have running.

The way you’ve been working with jobs so far has been much like when you were using Invoke-Command and specifying the name of a computer. Each time you contacted the computer, Invoke-Command created a new session. You’re doing much the same thing when you use Start-Job. With Invoke-Command, you were able to improve your efficiency by creating sessions. In the next section you’ll see how sessions work with jobs.

### 13.1.5 Running Jobs in Existing Sessions

Each background job runs in its own PowerShell session, paralleling the way each remote command is also executed in its own session. As was the case with remoting, this session can be a temporary one that exists only for the duration of the background job, or it can be run in an existing PSSession. But the way to do this isn’t obvious because the Start-Job cmdlet doesn’t have a -Session parameter. Instead, you have to use Invoke-Command with the -Session and -AsJob parameters. Here’s what that looks like. First, create a PSSession object:

PS> $s = New-PSSession -ComputerName W16DC01

Now pass that session object to Invoke-Command with -AsJob specified:

PS> $j = Invoke-Command -Session $s -ScriptBlock {$PID} -AsJob

The scriptblock that you’re passing in returns the process ID of the session. Use Receive-Job to retrieve it:

PS> Receive-Job $j

788

You can call Invoke-Command without -AsJob with the same session object and scriptblock:

PS> Invoke-Command -Session $s -ScriptBlock {$PID}

788

You get the same process ID back, which is expected because the session is persistently associated with the same process.

#### Start-Job and sessions

Why is there no -Session parameter on Start-Job? This parameter did exist at one point in the development of PowerShell v2. At that time, jobs and remoting used the same message transport. Using the same transport was found to be problematic for a number of reasons:

* It was inefficient for communication with local jobs.
* It required that the remoting service be enabled on the local machine, which has security implications.
* It required users to be running with admin privileges to be able to use the job feature.

To resolve these issues, the existing WS-MAN-based transport used by jobs was replaced with anonymous pipes. This change solved these problems, but it had the unfortunate side effect that jobs could no longer be directly run within PSSession instances because the PSSession object was tied to WS-MAN remoting.

Keep in mind that when a job is run in an existing PSSession, that session can’t be used to run additional tasks until the job has completed. This means you have to create multiple PSSession objects if you need to run multiple background tasks but want to avoid the overhead of creating new processes for each job. As always, it’s up to the script author to decide how best to manage resources for their script.

### 13.1.6 Job Types

If you look closely at the output from Get-Job in [sections 13.1.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-131-109.xhtml#ch13lev2sec2) through [13.1.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-131-109.xhtml#ch13lev2sec5) you’ll see that jobs have a PSJobTypeName property. You’ll notice that jobs run on the local machine have a PSJobTypeName of BackgroundJob, whereas jobs run on remote machines have RemoteJob.

There are other job types. The full list of job types is shown in [table 13.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-131-109.xhtml#ch13table02).

| **Table 13.2: PowerShell job types** | |
| --- | --- |
| **Job Type** | **Description** |
| BackgroundJob | Job created with Start-Job. |
| RemoteJob | Job created with -AsJob parameter of Invoke-Command. |
| PSWorkflowJob | Job created with -AsJob parameter of a workflow. See [section 13.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-132-110.xhtml#ch13lev1sec2). |
| PSScheduledJob | Job created by a scheduled job trigger. See [section 13.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-133-111.xhtml#ch13lev1sec3). |
| CIMJob | Job created with -AsJob parameter of a member of a CDXML module. |
| WMIJob | Job created with -AsJob parameter of a WMI cmdlet. Note the CIM cmdlets don’t have an -AsJob parameter. |
| PSEventJob | Job created by running Register-ObjectEvent and specifying an action with the -Action parameter. See [section 17.2.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17lev2sec5). |
| ConfigurationJob | Job created by Start-DSCconfiguration. Visible only if -Wait parameter of Start-DSCconfiguration is not used. |

Once a job has been created, it’s managed by the standard job cmdlets.

Using -AsJob with Get-WmiObject will return a WMIJob type if the target is the local machine or a remote machine:

PS> Get-WmiObject -Class Win32\_ComputerSystem -AsJob

PS> Get-WmiObject -Class Win32\_ComputerSystem -AsJob -ComputerName W16DC01

Get-Job will always show the job type (PSJobTypeName) in the default output:

PS> Get-Job

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

1 Job1 RemoteJob Failed True localhost,... Get-Date

5 Job5 RemoteJob Completed False W16DC01 $PID

8 Job8 WmiJob Completed True localhost Get-Wm...

10 Job10 WmiJob Completed True W16DC01 Get-Wm...

14 Job14 PSWorkflowJob Completed True localhost hi

Unfortunately, Get-Job doesn’t have a parameter that allows you to filter on job type, so you need to do something like this:

Get-Job | where PSJobTypeName -eq 'WmiJob'

or use whichever job type you’re interested in.

Now that you understand how jobs work, let’s look at combining jobs with workflows, as we promised in [chapter 12](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-12-102.xhtml#ch12).

## 13.2 Workflows as Jobs

We stated at the top of the chapter that workflows could be used in situations where the execution of the process needed to be paused. This pause could include the reboot of a remote machine or even the reboot of the machine on which the workflow is running. Making a workflow pause and then resume is dependent on checkpointing the workflow (persisting the data and execution state to disk) and then reading that data to restart the workflow. Dealing with a reboot is an extension of that approach. The first thing is to understand workflow checkpoints.

### 13.2.1 Checkpoints

The simplest way to checkpoint a workflow is to use the Checkpoint-Workflow activity:

workflow test-cw1

{

foreach ($i in 1..10) {$i}

Checkpoint-Workflow

foreach ($j in 50..60){$j}

}

test-cw1

Running test-cw1 you’ll see the numbers 1–10 and then 50–60 listed with no apparent break. You won’t be able to find any persisted data because workflows remove their persisted data on completion—unless they’re run as a job, in which case the persisted data is removed when the job is deleted.

#### Note

Checkpoint-Workflow doesn’t have any parameters. You have no control over how the checkpoint is performed or where the data is stored.

Checkpoint-Workflow can be used after any activity but not inside an InlineScript block. It takes an immediate checkpoint. When using checkpoints make sure of the following:

* The time taken to rerun the section you’ve checkpointed is longer (preferably a lot longer) than the time it takes to write the checkpoint to disk.
* You take checkpoints after critical steps so the workflow can be resumed rather than restarted.
* You take a checkpoint after steps that aren’t idempotent
* If your activity is in a pipeline and it’s checkpointed, the checkpoint doesn’t apply until the pipeline completes.
* Within parallel blocks the checkpoint doesn’t apply until the parallel processing has been applied to all items.
* In a sequence block checkpoints are applied after each activity.

You can create a checkpoint in three other ways. The first way is to use the –PSPersist workflow parameter:

workflow test-cw2 {

Get-Process

Get-Service

}

test-cw2 -PSPersist $true

In PowerShell v3 -PSPersist was a switch parameter. It’s now a Boolean with three possible states:

* **Default—** The -PSPersist parameter is not used. A checkpoint is taken at the beginning and end of the workflow together with any checkpoints explicitly created in the workflow.
* **$true—** Adds a checkpoint to the beginning and end of the workflow and after every activity, in addition to any checkpoints explicitly created in the workflow.
* **$false—** Adds no checkpoints. Only those checkpoints explicitly created in the workflow are taken.

Your second option is to use the –PSPersist activity parameter:

workflow test-cw3 {

Get-Process -PSPersist $true

Get-Service -PSPersist $true

}

test-cw3

When used at the activity level, setting –PSPersist to $true (the parameter was a switch in PowerShell v3) causes a checkpoint to be taken after the activity has completed. A value of $false, or if the parameter isn’t present, means a checkpoint won’t be taken.

The third option is to use the $PSPersistPreference preference variable:

workflow test-cw4 {

$PSPersistPreference = $true

Get-Process

Get-Service

$PSPersistPreference = $false

}

test-cw4

Setting $PSPersistPreference to $true causes a checkpoint to be taken after every activity until it’s set back to $false or the workflow ends.

Now that you know how to take checkpoints, how can you use them? You need to run your workflow as a job:

workflow test-cw5 {

foreach ($b in 1..1000) {

$b

Checkpoint-Workflow

}

}

test-cw5 -AsJob

The workflow will count from 1 to 1000, taking a checkpoint after each value. Count slowly to five and then shut down PowerShell. That’s right! Click the cross in the top-right corner and shut down the console.

Now open another PowerShell console that’s running with elevated privileges and use Get-Job. You’ll find your job has been suspended:

PS> Get-Job

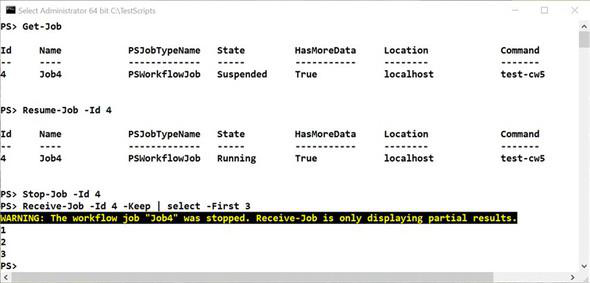
Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

5 Job4 PSWorkflowJob Suspended True localhost test-cw5

You can use Resume-Job to restart the workflow. Allow it to run for a few seconds and then stop the workflow job with Stop-Job. You can use Receive-Job to get the data back from the job. You’ll receive the warning shown in [figure 13.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-132-110.xhtml#ch13fig03) because you’ve stopped the job before completion.

Figure 13.3: Commands to resume a checkpointed workflow



[Figure 13.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-132-110.xhtml#ch13fig03) illustrates the sequence of commands required to restart a workflow. If you run

PS> $count = Receive-Job -Id 4 -Keep

PS> $count.Length

you’ll be able to check that your results are contiguous—there are no breaks in the number sequence, showing that the data was protected even though the workflow was forcibly stopped.

#### Note

If your workflow has output data prior to being checkpointed, that data won’t be present when you use Receive-Job.

This is too good a technique to be restricted to inadvertent workflow suspension. You need to be able to force a workflow into suspension as well.

### 13.2.2 Suspending Workflows

Deliberately suspending a workflow is a simple matter of using the Suspend-Workflow activity:

workflow test-ws1 {

Get-Process

Suspend-Workflow

Get-Service

}

test-ws1

The process information will be displayed and then the workflow will checkpoint and suspend. You can view the jobs with Get-Job and resume the job with Resume-Job. The data from the Get-Service activity can be viewed using Receive-Job. If you don’t want to see any results during the running of workflow, run it as a job:

workflow test-ws2 {

Get-Process -PSPersist $true

Suspend-Workflow

Get-Service

}

test-ws2 -AsJob -JobName swtest

The process data will be saved to disk with the workflow state, and once the workflow job has been resumed and run to completion you can view all your data. Saving the data until execution is complete is a good technique for long-running (overnight?) workflows.

You can use another PowerShell session to resume the job if required, as discussed in the previous section. It’s possible to use the Suspend-Job cmdlet to force the suspension of a workflow that’s running as a job. You’ll need to ensure that you’ve checkpointed the workflow if required so that your data is protected. You’ll also need to be able to get progress reports from the job so that you know when to perform the suspension. It’s usually easier to suspend the workflow from within the workflow using Suspend-Workflow.

#### Note

Suspend-Job and Resume-Job work only on workflow jobs.

You can also get your workflows to suspend automatically if there’s a terminating error within the workflow:

workflow test-ws3 {

Get-Process

Get-Service

}

test-ws3 -ErrorAction Suspend

Any errors that occur in the workflow will cause the workflow to suspend. Again, think about checkpointing to protect your data.

Suspending workflows solves the problem of making workflows interruptible, but we also said that workflows can survive reboots.

### 13.2.3 Workflows and Reboots

No, it’s not a role-playing game—in this section we’ll show you how workflows can survive a reboot. There are two scenarios you need to understand: a reboot on the target remote machine and a reboot on the local machine on which the workflow is executing. Let’s start with the easier case where you need to reboot the remote machine you’ve targeted with your workflow.

A machine can be rebooted using Restart-Computer. The following example retrieves the last time a machine was rebooted, forces a reboot, and then waits for the machine to become available before fetching the last boot time again:

workflow test-restart {

Get-CimInstance -ClassName Win32\_OperatingSystem |

Select-Object -ExpandProperty LastBootupTime

Restart-Computer -Wait

Get-CimInstance -ClassName Win32\_OperatingSystem |

Select-Object -ExpandProperty LastBootupTime

}

test-restart -PSComputerName W16TGT01

The key to pausing the workflow while the remoted machine reboots is the –Wait parameter on Restart-Computer. Using –Wait suppresses the PowerShell prompt and blocks the pipeline until all machines involved in the process have rebooted.

#### Waiting for restart

The –Wait parameter on Restart-Computer will cause your code to wait indefinitely for the remote machine to reboot (it has no effect when rebooting the local machine). You can modify the action of –Wait with the –For parameter, which causes the code to wait for a specific service or feature to be available. Here are your options:

* Default—Waits for Windows PowerShell to restart
* PowerShell—Can run commands in a Windows PowerShell remote session on the computer
* WMI—Receives a reply to a Win32\_ComputerSystem query for the computer
* WinRM—Can establish a remote session to the computer by using WS-Management

Alternatively, you can opt for a time delay by using the –TimeOut parameter, which enables you specify a number of seconds to wait for the remote machine to respond.

[Figure 13.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-132-110.xhtml#ch13fig04) shows this workflow in progress

Figure 13.4: Workflow paused while waiting for a remote machine to restart



While Restart-Computer is waiting for the remote machine to respond, a progress bar is shown, as can be seen in [figure 13.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-132-110.xhtml#ch13fig04). The messages on the progress bar indicate the following:

* Waiting for the restart to begin
* Verifying the computer has restarted
* Waiting for WMI connectivity
* Waiting for PowerShell connectivity
* Waiting for WinRM connectivity

These states correspond to the options available on the –For parameter of Restart-Computer. [Figure 13.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-132-110.xhtml#ch13fig05) shows the results of the workflow post completion.

Figure 13.5: Workflow restarting a remote computer



If you want to save the output or suppress the progress bar, run the workflow as a job.

You have two options when rebooting the local machine:

* Resume the workflow manually
* Resume the workflow automatically

In both cases a suspended job is produced when the reboot occurs. This job then has to be resumed either manually or via a scheduled job. Let’s start with the manual resumption. The previous workflow is modified to produce this code:

workflow test-restart {

Get-CimInstance -ClassName Win32\_OperatingSystem |

Select-Object -ExpandProperty LastBootupTime

Restart-Computer

Suspend-Workflow

Get-CimInstance -ClassName Win32\_OperatingSystem |

Select-Object -ExpandProperty LastBootupTime

}

The –Wait parameter has been removed from Restart-Computer (it doesn’t work when applied to the local machine), and a Suspend-Workflow activity is added immediately after the restart command. If you don’t force the suspension of the workflow, it will carry through and complete the tasks.

Run the workflow:

PS> test-restart -AsJob -JobName boottime

Once the machine has rebooted, open an elevated PowerShell console and run

PS> Resume-Job -Name boottime

When the job has completed, run the following to view the results:

PS> Receive-Job -Name boottime

09 May 2017 09:56:44

09 May 2017 14:19:39

The more complicated case is to create a situation where the workflow is automatically restarted after the reboot. This involves adding a scheduled task (the documentation says you can use a scheduled job, but that doesn’t seem to work) to the script.

#### Listing 13.3: Automatically resuming workflow on reboot

workflow test-restart { 1

Get-CimInstance -ClassName Win32\_OperatingSystem |

Select-Object -ExpandProperty LastBootupTime

Restart-Computer

Suspend-Workflow

Get-CimInstance -ClassName Win32\_OperatingSystem |

Select-Object -ExpandProperty LastBootupTime

}

$actionscript = '-NonInteractive -WindowStyle Normal –NoLogo

https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/images/enter6.jpg -NoProfile -NoExit

https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/images/enter7.jpg -Command "& {Get-Job -Name boottime | Resume-Job}"'

$pstart = "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" 2

Get-ScheduledTask -TaskName ResumeWF | 3

Unregister-ScheduledTask -Confirm:$false

$act = New-ScheduledTaskAction -Execute $pstart ` 4

-Argument $actionscript

$trig = New-ScheduledTaskTrigger -AtLogOn 5

Register-ScheduledTask -TaskName ResumeWF -Action $act ` 6

-Trigger $trig -RunLevel Highest

test-restart -AsJob -JobName boottime 7

* 1 Define workflow
* 2 Create PowerShell startup commands
* 3 Remove scheduled task
* 4 Create task action
* 5 Create trigger
* 6 Create scheduled task
* 7 Execute workflow

You’ll use the same workflow 1 as when you performed the workflow resumption manually. The scheduled task will invoke PowerShell 2 and run the command Get-Job -Name boottime | Resume-Job.

Any scheduled tasks of the same name are removed 3. A task action is created to run the script you created (running PowerShell) 4, and a job trigger is created 5 to run the scheduled task at logon.

The scheduled task is registered (created) 6 using the trigger and actions previously defined. You can then run the workflow 7 as a job.

Your machine will reboot, and when you log on you’ll see a PowerShell console running. After a few seconds, open another PowerShell console and review the jobs. Retrieve the data from the completed workflow job:

PS> Receive-Job -Name boottime -Keep

09 May 2017 09:48:42

09 May 2017 14:29:24

If you need your workflow to manage reboots, it’s much easier if the remote machine is rebooting rather than the local machine. Our recommendation is to ensure you write your workflows in this manner.

The concludes our examination of using jobs with workflows. Our last topic for this chapter looks at how you can use the Windows Scheduler and PowerShell jobs utilizing the scheduled jobs cmdlets.

## 13.3 Scheduled Jobs

The ability to execute long-running tasks asynchronously is a great benefit—you don’t have to sit and watch them run, for one thing! The PowerShell jobs we’ve discussed so far have all involved a manual start. Wouldn’t it be easier and more convenient if you could schedule jobs to start automatically—in particular, the ones that you run in the middle of the night?

### 13.3.1 Creating Scheduled Jobs

PowerShell v3 introduced the capability to do that through the PSScheduledJob module. The module contains a number of cmdlets:

PS> Get-Command -Module PSScheduledJob | Format-Wide -Column 3

Add-JobTrigger Disable-JobTrigger Disable-ScheduledJob

Enable-JobTrigger Enable-ScheduledJob Get-JobTrigger

Get-ScheduledJob Get-ScheduledJobOption New-JobTrigger

New-ScheduledJobOption Register-ScheduledJob Remove-JobTrigger

Set-JobTrigger Set-ScheduledJob Set-ScheduledJobOption

Unregister-ScheduledJob

The cmdlets show three objects involved in working with scheduled jobs:

* Scheduled jobs
* Scheduled job triggers
* Scheduled job options

We’ll examine each of these areas in this section.

#### Note

Scheduled jobs are different from scheduled tasks. A scheduled job runs a PowerShell job on a trigger activated by the Windows Task Scheduler. A scheduled task runs a script, or command, on activation of a Windows Task Scheduler trigger.

The major difference between scheduled jobs and the jobs discussed in the previous sections is that a scheduled job survives the closure of the PowerShell console, but a regular job doesn’t. A scheduled job will continue to exist and run, according to its schedule, until it’s disabled or deleted. You can manage the job objects produced by a scheduled job with the standard job cmdlets.

Like with everything in PowerShell, it’s easier to explain scheduled jobs with examples. You’ll start by creating a simple job that will run a number of times. First, you need a trigger:

PS> $t = New-JobTrigger -Once -At "09/05/2017 15:10" `

-RepetitionInterval (New-TimeSpan -Minutes 1) `

-RepetitionDuration (New-TimeSpan -Minutes 10)

This trigger will execute on 9 May 2017 at 15:10 (3:10 p.m.). It will repeat execution every minute for 10 minutes. The -At parameter specifies the start time. When using -Once with -At ensure that the start time is in the future.

Many other options are available for setting triggers:

* -AtLogOn—Starts the scheduled job when a specified user, or users, logs on to the machine.
* -AtStartUp—Starts the scheduled job when Windows starts.
* -Daily—Specifies a recurring job that runs every day. You set a number of days between jobs being executed using the -DaysInterval parameter.
* -Weekly—Specifies a job that’s run weekly. Use the -DaysOfWeek parameter to control the days it runs—for instance, you may want the job to run only on Saturdays and Sundays. You can specify a number of weeks between executions using -WeeksInterval.

Once you have the trigger defined, you can create the scheduled job:

PS> Register-ScheduledJob -Name PiASJ1 -ScriptBlock {Get-Process} `

-Trigger $t -RunNow

Id Name JobTriggers Command Enabled

-- ---- ----------- ------- -------

1 PiASJ1 1 Get-Process True

You specify the job’s name, a scriptblock for the job to execute, and the trigger. You can use the -RunNow parameter of Register-ScheduledJob to run the job once as you register it. This is a useful test to ensure that everything works correctly.

#### Note

Set-SetScheduledJob also has a -RunNow parameter for immediate execution of a scheduled job.

Scheduled jobs can be found in the Task Scheduler at Library\Microsoft\Windows\PowerShell\ScheduledJobs.

Your scheduled jobs and their results are stored in $home\AppData\Local\Microsoft\Windows\PowerShell\ScheduledJobs. Look in the <jobname>\Output folder for the results.

By default, the results of 32 instances of each scheduled job are stored. Older jobs will be overwritten as necessary. You can modify the number of saved instances using -MaxResultCount.

You can access the results of your scheduled job using Get-Job:

PS> Get-Job -Name PiASJ1 | Format-Table -AutoSize

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

3 PiASJ1 PSScheduledJob Completed True localhost Get-Process

4 PiASJ1 PSScheduledJob Completed True localhost Get-Process

5 PiASJ1 PSScheduledJob Completed True localhost Get-Process

6 PiASJ1 PSScheduledJob Completed True localhost Get-Process

7 PiASJ1 PSScheduledJob Completed True localhost Get-Process

8 PiASJ1 PSScheduledJob Completed True localhost Get-Process

9 PiASJ1 PSScheduledJob Completed True localhost Get-Process

10 PiASJ1 PSScheduledJob Completed True localhost Get-Process

11 PiASJ1 PSScheduledJob Completed True localhost Get-Process

12 PiASJ1 PSScheduledJob Completed True localhost Get-Process

You can access the job’s results using Receive-Job:

PS> Receive-Job -Id 6

The -Keep parameter hasn’t been used, and the results will appear to have been stripped out of the job object. In reality, this isn’t the case. The data is still available on disk and can be accessed in another PowerShell console. If you use the -Keep parameter, you’ll be able to repeatedly access the data in the same console.

A scheduled job can be started outside its schedule using Start-Job:

PS> Start-Job -DefinitionName PiASJ1

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

13 PiASJ1 PSScheduledJob Running True localhost Get-Process

The job is given a type of scheduled job, but the output isn’t persisted to disk and will be lost when the console is closed.

### 13.3.2 Modifying a Scheduled Job

Once you’ve created a scheduled job, you can modify a number of features. Let’s start with the trigger.

#### Changing a Trigger

A scheduled job can have more than one trigger. You might want to run the scheduled job at 10 a.m. on a Monday and 5 p.m. on a Friday, for instance. To add a trigger to an existing scheduled job, first define the trigger:

PS> $t2 = New-JobTrigger -Once -At "09/05/2016 18:30" `

-RepetitionInterval (New-TimeSpan -Minutes 1) `

-RepetitionDuration (New-TimeSpan -Minutes 10)

then add the trigger:

PS> Add-JobTrigger -Trigger $t2 -Name PiASJ1

A scheduled job’s triggers can be viewed:

PS> Get-JobTrigger -Name PiASJ1

Id Frequency Time DaysOfWeek Enabled

-- --------- ---- ---------- -------

1 Once 09/05/2017 15:10:00 True

2 Once 09/05/2017 18:30:00 True

Old triggers can be removed:

PS> Remove-JobTrigger -Name PiASJ1 -TriggerId 1

#### Scheduled Job Options

A number of options are available to you when creating a scheduled job. The default options are listed here:

PS> New-ScheduledJobOption

StartIfOnBatteries : False

StopIfGoingOnBatteries : True

WakeToRun : False

StartIfNotIdle : True

StopIfGoingOffIdle : False

RestartOnIdleResume : False

IdleDuration : 00:10:00

IdleTimeout : 01:00:00

ShowInTaskScheduler : True

RunElevated : False

RunWithoutNetwork : True

DoNotAllowDemandStart : False

MultipleInstancePolicy : IgnoreNew

JobDefinition :

You can control how the scheduled job behaves if the machine is on, or goes on, battery power. There are a number of options detailing the job’s response to the machine being idle or not. You can configure the job to be run with elevated privileges and even hide the job from the Task Scheduler GUI.

RunElevated is probably the most important option. If you use this, you need to also specify the -Credential option in Register-ScheduledJob or Set-ScheduledJob.

### 13.3.3 Managing Scheduled Jobs

You can view the instances of a scheduled job:

PS> Get-job -Name PiASJ1

The latest instances can be viewed:

PS> Get-job -Name PiASJ1 -Newest 2

Jobs executed in a particular time interval can be found:

PS> Get-job -Name PiASJ1 -Before "09/05/2017 15:20:00" `

-After "09/05/2017 15:15:00"

Individual instances can be removed:

PS> Remove-Job -Id 12

or you can remove all instances:

PS> Remove-Job -Name PiASJ1

The scheduled job itself can be removed:

PS> Unregister-ScheduledJob -Name PiASJ1

This also removes the data stored on disk.

## 13.4 Summary

* PowerShell jobs run asynchronously.
* A background job runs in a new PowerShell process. Other jobs execute in process but on separate threads.
* Start-Job creates and runs a new background job.
* Use -Keep with Receive-Job to ensure the data remains accessible.
* Wait-Job blocks further interactive processing until the job, or jobs, completes.
* Remove-Job deletes one, many, or all jobs present on the system.
* Jobs are deleted when the PowerShell console is closed.
* You can run jobs on the local and/or remote machines.
* A job consists of a parent job and one or more child jobs. One child job is created per remote machine specified to Invoke-Command when using the -AsJob parameter.
* Multiple job types exist. They are started in different ways but are all managed using the standard job cmdlets.
* Workflow state and data can be explicitly persisted using the Checkpoint-Workflow activity or automatically after each activity using the -PSPersist parameter. Be aware that persisting a workflow is expensive, so this feature should be used when needed rather than enabling it all the time.
* Suspend-Workflow will persist a workflow and halt execution. A suspended job is created that can be resumed at a later time.
* The job cmdlets can be used to manage workflow jobs.
* Resume-Job will restart a suspended workflow job.
* PowerShell workflows can survive a reboot of the remote or local machine.
* A reboot of the local machine while a workflow is executing can be managed manually or automatically using a scheduled task.
* Scheduled jobs are run via the Windows Task Scheduler.
* The results from a scheduled job are stored on disk and are available between PowerShell sessions.
* Scheduled job triggers can be defined on a time basis, user logon, or machine startup.
* The instances of a scheduled job are managed with the standard job cmdlets.
* Job triggers can be added and removed.
* Scheduled job options include running with elevated privileges and hiding from the Task Scheduler.
* Scheduled job instances can be removed singly or in bulk.

Workflows ([chapter 12](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-12-102.xhtml#ch12)) and jobs (this chapter) provide a strong foundation for performing tasks in a production environment. When writing production code, you need to be able to manage errors that may occur when your code is running—as you’ll see in the next chapter.

## Chapter 14: Errors and Exceptions

### Overview

This chapter covers

* Error handling
* Dealing with terminating errors
* Working with event logs

*Progress, far from consisting in change, depends on retentiveness. Those who cannot remember the past are condemned to repeat it.  
George Santayana, The Life of Reason*

It’s always useful to keep in mind that PowerShell isn’t merely a shell or scripting language. Its primary purpose is to be an automation tool and perform critical management tasks on a server, such as send software updates, inspect log files, or provision user accounts. You need to be sure that either the task is completed properly or the reason for failure is appropriately recorded.

In this chapter, we’ll focus on how PowerShell reports, records, and manages error conditions. Handling of error conditions is one of the areas where PowerShell shines compared to other scripting tools. The support for diagnostic tracing and logging is practically unprecedented in traditional scripting languages. Unfortunately, these features don’t come entirely free—there are costs in terms of complexity and execution overhead that aren’t there in other environments. All these capabilities are a part of PowerShell as a management tool; Microsoft set a higher bar for PowerShell than has been set for most other language environments.

We’ll begin by looking at the error processing subsystem. Errors in PowerShell aren’t error codes, strings, or even exceptions as found in languages such as C# and VB.NET. They’re rich objects that include almost everything you could think of that might be useful in debugging a problem.

**Note**

Some people dislike (okay, despise) the use of the word rich in this context. But given the wealth of information that PowerShell error objects contain, rich is the right word.

We’ll examine these ErrorRecord objects in detail, along with how they’re used by the various PowerShell mechanisms to manage error conditions.

## 14.1 Error Handling

Error handling in PowerShell is structured. PowerShell errors aren’t bits of text written to the screen—they’re rich objects that contain a wealth of information about where the error occurred and why. There’s one aspect to error handling in PowerShell that’s unique: the notion of terminating versus nonterminating errors. This aspect aligns with the streaming model that PowerShell uses to process objects.

Here’s a simple example that will help you understand this concept. Think about how removing a list of files from your system should work. You stream this list of files to the cmdlet that will delete the files. But imagine that you can’t delete all the files on the list for various reasons. Do you want the command to stop processing as soon as it hits the first element in the list? The answer is probably no. You’d like the cmdlet to do as much work as it can but capture any errors so that you can look at them later. This is the concept of a nonterminating error—the error is recorded and the operation continues. There are times when you do want an operation to stop on the first error. These are called terminating errors. Sometimes you want an error to be terminating in one situation and nonterminating in another, and PowerShell provides mechanisms that allow you to do that.

#### Note

PowerShell is based on .NET, but you need to be aware that in .NET errors are all terminating unless they’re handled somewhere in the calling code. PowerShell cmdlets, and advanced functions or scripts, introduce the concept of nonterminating errors, meaning that the error has been managed internally.

Because the architecture supports multiple nonterminating errors being generated by a pipeline, it can’t just throw or return an error. Here’s where streaming comes into play: nonterminating errors are written to the error stream. By default, these errors are displayed, but there are a number of other ways of working with them. In the next few sections, we’ll look at those mechanisms. First, we need to look at the error records themselves.

### 14.1.1 ErrorRecords and the Error Stream

As we delve into the topic of error handling, we’ll first look at capturing error records in a file using redirection, and then you’ll learn how to capture error messages in a variable. By capturing these errors instead of merely displaying them, you can go back to analyze and hopefully fix what went wrong.

First, let’s review the normal behavior of objects in the pipeline. Output objects flow from cmdlet to cmdlet, but error records are written directly to the default output processor. By default, this is the Out-Default cmdlet, and the error records are displayed:

PS> Get-ChildItem -Path nosuchfile

Get-ChildItem : Cannot find path 'C:\test\nosuchfile' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nosuchfile

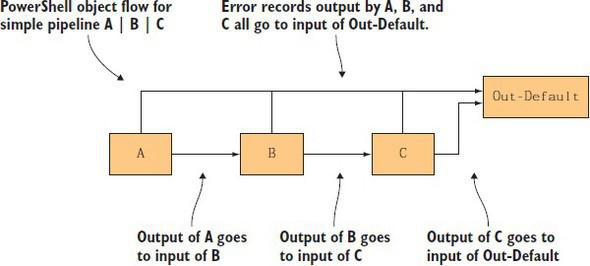
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound: (C:\test\nosuchfile:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

These flows are shown in [figure 14.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-141-114.xhtml#ch14fig01).

Figure 14.1: This diagram shows the output object and error record routing; then, the simple pipeline A | B | C is run from a PowerShell host process like PowerShell.exe or PowerShell\_ISE.exe. Output objects go to the next command in the pipeline, and error objects go directly to Out-Default



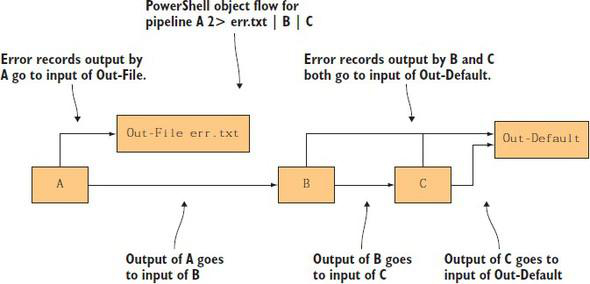
In [figure 14.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-141-114.xhtml#ch14fig01), you see the output objects go from A to B to C and finally to Out-Default. But the error record streams are all merged and go directly to Out-Default.

When you use the redirection operators discussed in [chapter 4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-4-32.xhtml#ch04), you can change flow. For example, you can redirect the error messages to a file:

PS> Get-ChildItem -Path nosuchfile 2> err.txt

This changes the process to look like what’s shown in [figure 14.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-141-114.xhtml#ch14fig02).

Figure 14.2: Revised pipeline including the use of redirection operators



This approach has the downside that the error message is rendered to displayable text before writing it to the file. When that happens, you lose all the extra information in the objects. Look at what was saved to the file:

PS> Get-Content .\err.txt

Get-ChildItem : Cannot find path 'C:\test\nosuchfile' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nosuchfile 2> err.txt

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

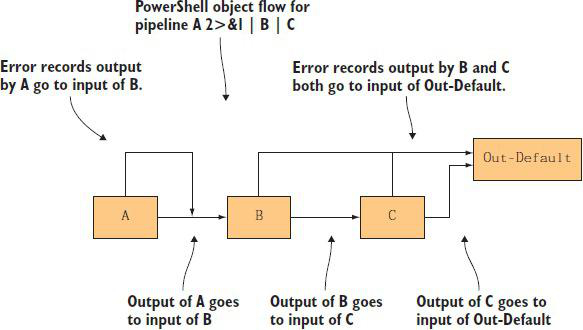
+ CategoryInfo : ObjectNotFound: (C:\test\nosuchfile:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

The error text is there as it would’ve been displayed on the console, but you’ve lost all the elements of the object that haven’t been displayed. This lost information may be critical to diagnosing the problem. You need a better way to capture this information. The first mechanism we’ll look at is capturing the error records by using the stream merge operator 2>&1 and then assigning the result to a variable.

When you add error stream merging to the picture, the flow of objects changes. With stream merging, instead of having all error records going to the default output stream, they’re routed into the output stream, and the combined set of objects is passed to the input of the next command. This flow is shown in [figure 14.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-141-114.xhtml#ch14fig03).

Figure 14.3: Revised pipeline including the addition of error stream merging



Let’s see how this works. First, use the stream merge operator to capture the error stream in a variable by using assignment:

PS> $err = Get-ChildItem -Path nosuchfile 2>&1

You can use Get-Member to display the properties on the object. Use the -Type parameter on Get-Member to filter the display and only show the properties:

PS> $err | Get-Member -Type property

Some of the property names are a little tricky to figure out, so further explanation is in order. [Table 14.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-141-114.xhtml#ch14table01) lists the important properties, their types, and a description of each.

| **Table 14.1: ErrorRecord properties and their descriptions** | | |
| --- | --- | --- |
| **Property name** | **Property type** | **Description** |
| CategoryInfo | ErrorCategoryInfo | This string breaks errors into a number of broad categories. |
| ErrorDetails | ErrorDetails | This may be null. If present, ErrorDetails can specify additional information, most importantly ErrorDetails.Message, which (if present) is a more exact description and should be displayed instead of Exception.Message. |
| Exception | System.Exception | This is the underlying .NET exception corresponding to the error that occurred. |
| FullyQualifiedErrorId | System.String | This identifies the error condition more specifically than either the ErrorCategory or the Exception. Use FullyQualifiedErrorId to filter highly specific error conditions. Note that this is a nonlocalized field, so performing string matches against it will continue to work regardless of language settings. |
| InvocationInfo | InvocationInfo | This object contains information about where the error occurred—typically the script name and line number. |
| TargetObject | System.Object | This is the object that was being operated on when the error occurred. It may be null, because not all errors will set this field. |

You can look at the values of an error record’s properties by piping the error object into Format-List. To see all the properties, you must specify –Property \* along with –Force. This command tells the formatting subsystem to skip the default presentation and show all properties. The result looks like this:

PS> $err | Format-List -Property \* -Force

writeErrorStream : True

PSMessageDetails :

Exception : System.Management.Automation.ItemNotFoundException:

Cannot find path 'C:\test\nosuchfile' because it

does not exist. at

System.Management.Automation.SessionStateInternal.

GetChildItems(String path, Boolean recurse,

UInt32 depth, CmdletProviderContext context)

At Microsoft.PowerShell.Commands.

GetChildItemCommand.ProcessRecord()

TargetObject : C:\test\nosuchfile

CategoryInfo : ObjectNotFound: (C:\test\nosuchfile:String)

[Get-ChildItem],ItemNotFoundException

FullyQualifiedErrorId :

PathNotFound,Microsoft.PowerShell.Commands.

GetChildItemCommand

ErrorDetails :

InvocationInfo : System.Management.Automation.InvocationInfo

ScriptStackTrace : at <ScriptBlock>, <No file>: line 1

PipelineIterationInfo : {0, 1}

In this output, you can see the exception that caused the error was ItemNotFoundException. The TargetObject property contains the full path the cmdlet used to locate the item. This overall error is placed in the broader category of ObjectNotFound. There are no additional error details for this object.

Let’s look closer at the InvocationInfo property. This member provides information about where the error occurred. Here’s what it looks like:

PS> $err.InvocationInfo

MyCommand : Get-ChildItem

BoundParameters : {}

UnboundArguments : {}

ScriptLineNumber : 1

OffsetInLine : 8

HistoryId : 8

ScriptName :

Line : $err = Get-ChildItem -Path nosuchfile 2>&1

PositionMessage : At line:1 char:8

+ $err = Get-ChildItem -Path nosuchfile 2>&1

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

PSScriptRoot :

PSCommandPath :

InvocationName : Get-ChildItem

PipelineLength : 0

PipelinePosition : 0

ExpectingInput : False

CommandOrigin : Internal

DisplayScriptPosition :

Because you enter this command on the command line, the script name is empty and the script line number is 1. OffsetInLine is the offset in the script line where the error occurred. Other information is also available, such as the number of commands in the pipeline that caused an error, as well as the index of this command in the pipeline. This message also includes the line of script text where the error occurred. Finally, there’s the PositionMessage property. This property takes all the other information and formats it into what you see in PowerShell errors.

Extracting all the detailed information from an error record is a fairly common occurrence when debugging scripts, so it’s worth writing a small helper function to make it easier. The next listing shows a function that will dump out all the properties of an error object and then iterate through any InnerException properties on the error record exception to show all the underlying errors that occurred.

#### Listing 14.1: The Show-ErrorDetails function

function Show-ErrorDetails

{

param(

$ErrorRecord = $Error[0]

)

$ErrorRecord | Format-List -Property \* -Force

$ErrorRecord.InvocationInfo | Format-List -Property \*

$Exception = $ErrorRecord.Exception

for ($depth = 0; $Exception -ne $null; $depth++)

{ "$depth" \* 80 1

$Exception | Format-List -Property \* -Force 2

$Exception = $Exception.InnerException 3

}

}

* 1 Show depth of nested exception
* 2 Show exception properties
* 3 Link to nest exceptions

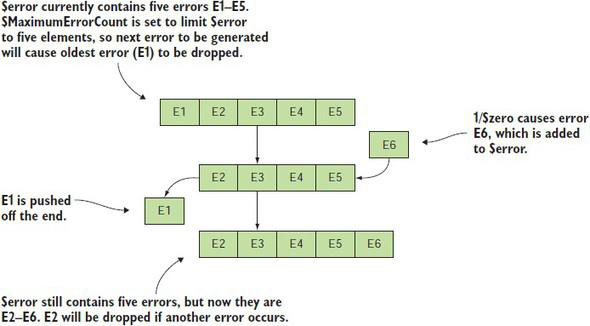
This function takes a single parameter that holds the error record to display. By default, it shows the most recent error recorded in $error. It begins by showing all the properties in the record followed by the invocation information for the faulting command. Then it loops, tracing through any nested exceptions 3, showing each one 2 proceeded by a separator 1 line showing the nesting depth of the displayed exception.

There’s a lot of information in these objects that can help you figure out where and why an error occurred. The trick is to make sure you have the right error objects available at the right time. It isn’t possible to record every error that occurs—it would take up too much space and be impossible to manage. If you limit the set of error objects that are preserved, you want to make sure that you keep those you care about—having the wrong error objects doesn’t help. Sometimes you’re interested only in certain types of errors or only in errors from specific parts of a script. To address these requirements, PowerShell provides a rich set of tools for capturing and managing errors.

### 14.1.2 The $error Variable and –ErrorVariable Parameter

The point of rich error objects is that you can examine them after the error has occurred and possibly take remedial action. To do that, you have to capture them first. In the previous section, we showed you how to redirect the error stream, but the problem with this approach is that you have to think of it beforehand. Because you don’t know when errors occur, in practice you’d have to do it all the time. Fortunately, PowerShell performs some of this work for you and automatically “remembers the past,” at least as far as errors go. There’s a special variable $error that contains a collection of the errors that occurred while the engine was running. This collection is maintained as a circular bounded buffer. As new errors occur, old ones are discarded, as shown in [figure 14.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-141-114.xhtml#ch14fig04).

Figure 14.4: How the $error variable handles new errors when MaximumErrorCount has been reached. The oldest error is dropped, and the new one is added to the end



The number of errors that is retained is controlled by the $MaximumErrorCount variable, which can be set to a number from 256 (the default setting) to 32768. The collection in $error is an array (technically an instance of System.Collections.ArrayList) that buffers errors as they occur. The most recent error is always stored in $error[0].

#### Note

Although it’s tempting to think that you could set $MaximumErrorCount to some large value (32768 is the largest allowed) and never have to worry about capturing errors, in practice this strategy isn’t a good idea. Rich error objects also imply fairly large error objects. If you set $MaximumErrorCount to too large a value, you won’t have any memory left. In practice, there’s usually no reason to set it to anything larger than the default, though you may set it to something smaller if you want to make more space available for other things. Also, even if you have only a few objects, these objects may be large. If you find that this is the case for a particular script, you can change the maximum error count to something small. As an alternative, you could clean out all the entries in $error by calling $error.Clear().

Let’s explore using the $error variable. You’ll start with the same error as before:

PS> Get-ChildItem -Path nosuchfile

Get-ChildItem : Cannot find path 'C:\test\nosuchfile' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nosuchfile

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound: (C:\test\nosuchfile:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

You didn’t explicitly capture it, but it’s available in $error[0] with all the error properties:

PS> $error[0]

Get-ChildItem : Cannot find path 'C:\test\nosuchfile' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nosuchfile

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound: (C:\test\nosuchfile:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

For example, here’s the exception object:

PS> $error[0].Exception

Cannot find path 'C:\test\nosuchfile' because it does not exist.

and here’s the target object that caused the error:

PS> $error[0].TargetObject

C:\test\nosuchfile

Now let’s do something that will cause a second error:

PS> 1/0

Attempted to divide by zero.

At line:1 char:1

+ 1/$null

+ ~~~~~~~

+ CategoryInfo : NotSpecified: (:) [], RuntimeException

+ FullyQualifiedErrorId : RuntimeException

Here you have a division-by-zero error.

Let’s verify that the second error is in $error[0]. Look at the exception member:

PS> $error[0].Exception

Attempted to divide by zero.

Yes, it is. You’ll also verify that the previous error, “file not found,” is now in position 1:

PS> $error[1].Exception

Cannot find path 'C:\test\nosuchfile' because it does not exist.

Again, yes, it is. As you can see, each new error shuffles the previous error down one element in the array.

#### Tip

The key lesson to take away from this is that when you’re going to try to diagnose an error, you should copy it to a “working” variable so it doesn’t get accidently shifted out from under you because you made a mistake in one of the commands you’re using to examine the error. In particular, you should never depend on the value of $error when writing tests because you may end up looking at the wrong value due to side effects in the test code. In general, you should consider $error as a trace log of errors that have occurred.

The $error variable is a convenient way to capture errors automatically, but there are two problems with it:

* $error captures only a limited number of errors—the default is 256—so important information may fall off the end of the buffer.
* $error contains all the errors that occur, regardless of where they came from or what command generated them, mixed together in a single collection. You’ll find it hard to locate the information you need to diagnose a specific problem.

You can work around the first problem by using redirection to capture all the errors, but that still doesn’t address mixing all the errors together. To deal with this second issue, when you want to capture all the errors from a specific command, you use a standard parameter available on all commands called -ErrorVariable. This parameter names a variable to use for capturing all the errors that the command generates—the command’s error handler performs the action of writing the information to the variable. Here’s an example: this command generates three error objects, because the files nofuss, nomuss, and nobother don’t exist:

PS> Get-ChildItem -Path nofuss, nomuss, nobother -ErrorVariable errs

Get-ChildItem : Cannot find path 'C:\test\nofuss' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nofuss, nomuss, nobother -ErrorVariable errs

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nofuss:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

Get-ChildItem : Cannot find path 'C:\test\nomuss' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nofuss, nomuss, nobother -ErrorVariable errs

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nomuss:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

Get-ChildItem : Cannot find path 'C:\test\nobother' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nofuss, nomuss, nobother -ErrorVariable errs

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nobother:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

In the command, you specified the name of the error variable to place these records into: errs.

#### Note

The argument to -ErrorVariable is the name of the variable with no leading $. If errs had been written as $errs, then the errors would’ve been stored in the variable named by the value in $errs, not $errs itself. Also note that the -ErrorVariable parameter works like a tee—the objects are captured in the variable, but they’re also streamed to the error output. If -ErrorAction is set to Ignore, the errors won’t be captured in the variable.

Let’s verify that the errors were captured. First, the number of elements in $err should be 3:

PS> $errs.Count

3

It is. Now dump the errors themselves:

PS> $errs

Get-ChildItem : Cannot find path 'C:\test\nofuss' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nofuss, nomuss, nobother -ErrorVariable errs

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nofuss:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

Get-ChildItem : Cannot find path 'C:\test\nomuss' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nofuss, nomuss, nobother -ErrorVariable errs

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nomuss:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

Get-ChildItem : Cannot find path 'C:\test\nobother' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path nofuss, nomuss, nobother -ErrorVariable errs

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nobother:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

They do, in fact, match the original error output.

#### Note

The errors should match the original output because they’re the same error objects. The -ErrorVariable parameter (alias –ev) captures references to each object written to the error stream. In effect, the same object is in two places at once—well, three if you count the default $error variable.

Because there’s no need to see the object twice, you can use redirection to discard the written objects and save only the references stored in the specified variable. Let’s rerun the example this way:

PS> Get-ChildItem -Path nofuss, nomuss, nobother `

-ErrorVariable errs 2>$null

This time nothing is displayed; verify the error count:

PS> $errs.Count

3

It’s 3 again, as intended. Let’s check the TargetObject member of the last error object to verify that it’s the filename nobother:

PS> $errs[2].TargetObject

C:\test\nobother

Yes, it is. This example illustrates a more sophisticated way of capturing error objects than merely displaying them. In section 14.1.5, you’ll see an even more flexible way to control how errors are redirected.

All of these mechanisms provide useful tools for handling collections of error objects, but sometimes all you care about is that an error occurred at all. A couple of additional status variables, $? and $LASTEXITCODE, enable you to determine whether an error occurred.

### 14.1.3 Determining Whether a Command Had an Error

Displaying errors is useful—it lets the user know what happened. But scripts also need to know when an error has occurred so they can react properly. For example, a script shouldn’t try to remove a file if the cd into the directory containing the file failed. PowerShell makes this easy by providing two error variables ($? and $LASTEXITCODE) that capture the command status. First, to see if an error occurred when executing a command, a script can check the status of the variable $?, a simple Boolean variable that holds the execution status of the last command.

#### Note

The use of the $? variable is borrowed from the UNIX shells.

The $? variable will be true if the entire operation succeeded and false otherwise. If any of the operations wrote an error object, then $? will be set to false even if the error was discarded using redirection. This is an important point: it means that a script can determine whether an error occurred even if the error isn’t displayed. Here are examples showing the use of $?. First, you call Get-Item, passing in items you know exist and don’t exist:

PS> Get-Item c:, nosuchfile, c:

Directory: C:\

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 02/05/2016 14:52 test

Get-Item : Cannot find path 'C:\test\nosuchfile' because it does not exist.

At line:1 char:1

+ Get-Item c:, nosuchfile, c:

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nosuchfile:String) [Get-Item], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetItemCommand

d----- 02/05/2016 14:52 test

You get the expected error:

PS> $?

False

and $? is false. Now try the same command, but this time specify only items that exist:

PS> Get-Item c:, c:

Directory: C:\

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 02/05/2016 14:52 test

d----- 02/05/2016 14:52 test

PS> $?

True

This time there are no errors, and $? is true.

Whereas the $? variable only indicates success or failure, the second error variable $LASTEXITCODE contains the exit code of the last command run. But this applies to only two types of commands: native or external commands and PowerShell scripts (but not functions).

#### Note

On Windows, when a process exits it can return a single integer as its exit code. This integer is used to encode a variety of conditions, but the only one we’re interested in is whether it’s zero or non-zero. This convention is used by almost all programs. If they were successful, then their exit code is zero. If they encountered an error, then the exit code is non-zero.

PowerShell captures the exit code from a script or executable in $LASTEXITCODE, and if that value is non-zero, it sets $? to false. Let’s use cmd.exe to demonstrate this. You can tell cmd.exe to execute a single command by passing it the /c option along with the text of the command. In this example, the command you want to run is exit, which takes a value to use as the exit code for the command:

PS> cmd /c exit 0

You told cmd to exit with code 0. Verify this by checking the values of $? and $LASTEXITCODE, respectively:

PS> $?

True

PS> $LASTEXITCODE

0

As expected, the exit code was zero, and consequently $? is true.

#### Note

Ensure you access the value of $? first because accessing $LASTEXITCODE will reset $?.

Next, try it with a non-zero value:

PS> cmd /c exit 1

PS> $?

False

PS> $LASTEXITCODE

1

This time the exit code is 1, so $? is set to false. You can do the same exercises with scripts. First, create a script that exits with a zero exit code:

PS> 'exit 0' > invoke-exit.ps1

PS> .\invoke-exit.ps1

PS> $?

True

PS> $LASTEXITCODE

0

$LASTEXITCODE is 0, and $? is true. Now try it with a non-zero value:

PS> 'exit 25' > invoke-exit.ps1

PS> .\invoke-exit.ps1

PS> $?

False

PS> $LASTEXITCODE

25

Now $LASTEXITCODE contains the value the script exited with, which is 25, and $? is set to false.

So far, we’ve looked at how to capture errors and how to detect when they occur. Next, we’ll explore some of the methods PowerShell provides to control what happens when an error is generated.

### 14.1.4 Controlling the Actions Taken on an Error

Earlier, we talked about the differences between terminating and nonterminating errors. Sometimes you want to be able to turn nonterminating errors into terminating ones because the operation you’re performing is too critical to tolerate nonterminating errors. Imagine you’re setting up a website for a user. You want to reuse a directory that had been previously used for someone else. First, you want to remove all the old files and then install the new user’s files. Obviously, you can’t start installing the new files until all the old ones are deleted. In this situation, the failure to delete a file, which is normally a nonterminating error, must now be treated as a terminating error. The next step in the process can’t begin until the current step is 100% complete.

The way to control whether errors are treated as terminating or nonterminating is by setting the error action policy, which you do by setting the error action preference. This is a mechanism that allows you to control the behavior of the system when an error occurs. There are a number possible settings for this preference as described in [table 14.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-141-114.xhtml#ch14table02).

| **Table 14.2: The supported identifiers and numeric equivalents for ErrorActionPreference and the -ErrorAction common parameter** | | |
| --- | --- | --- |
| **Identifier** | **Numeric value** | **Descriptions** |
| Continue | 2 | This is the default preference setting. The error object is written to the output pipe and added to $error, and $? is set to false. Execution then continues at the next script line. |
| SilentlyContinue | 0 | When this action preference is set, the error message isn’t written to the output pipe before continuing execution. Note that it’s still added to $error, and $? is still set to false. Again, execution continues at the next line. |
| Stop | 1 | This error action preference wraps a nonterminating error as a terminating error. The error object is then thrown as an exception instead of being written to the output pipe. $error and $? are still updated. Execution does not continue. |
| Inquire | 3 | Prompts the user requesting confirmation before continuing with the operation. At the prompt, the user can choose to continue, stop, or suspend the operation. |
| Ignore | 4 | Ignores the error and continues processing. Works with the -ErrorAction common parameter but isn’t allowed as an option for $ErrorActionPreference. |
| Suspend | 5 | Suspends the command for further investigation. Works only with PowerShell workflows. |

There are two ways to set the error action preference: by setting the $ErrorActionPreference variable as in

PS> $ErrorActionPreference = 'SilentlyContinue'

or by using the -ErrorAction (or -ea) parameter that’s available on all cmdlets, advanced functions, and advanced scripts.

Let’s see examples of these preferences in action. Here’s a simple one. First, run a command that has some nonterminating errors. You’ll use the Get-Item cmdlet to get two items that exist and two items that don’t exist:

PS> Get-Item -Path c:\, nosuchfile, c:\test, c:\nosuchfolder

Directory:

Mode LastWriteTime Length Name

---- ------------- ------ ----

d--hs- 02/05/2016 09:51 C:\

Get-Item : Cannot find path 'C:\test\nosuchfile' because it does not exist.

At line:1 char:1

+ Get-Item -Path c:\, nosuchfile, c:\test, c:\nosuchfolder

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nosuchfile:String) [Get-Item], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetItemCommand

Directory: C:\

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 02/05/2016 15:48 test

Get-Item : Cannot find path 'C:\nosuchfolder' because it does not exist.

At line:1 char:1

+ Get-Item -Path c:\, nosuchfile, c:\test, c:\nosuchfolder

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\nosuchfolder:String) [Get-Item], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetItemCommand

When you look at the output, you can see that there are two output objects and two error messages. You can use redirection to discard the error messages, making the code easier to read:

PS> Get-Item -Path c:\, nosuchfile, c:\test, c:\nosuchfolder 2> $null

Directory:

Mode LastWriteTime Length Name

---- ------------- ------ ----

d--hs- 02/05/2016 09:51 C:\

Directory: C:\

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 02/05/2016 15:48 test

Now you only see the output objects because you’ve sent the error objects to $null. You can use the -ErrorAction parameter to do the same:

PS> Get-Item -Path c:\, nosuchfile, c:\test, c:\nosuchfolder `

-ErrorAction SilentlyContinue

Directory:

Mode LastWriteTime Length Name

---- ------------- ------ ----

d--hs- 02/05/2016 09:51 C:\

Directory: C:\

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 02/05/2016 15:48 test

Again, the error messages aren’t displayed, but this time it’s because they aren’t being written to the console. Instead of being written and discarded the errors will be written to $error. If you use -ErrorAction Ignore, the errors are ignored and discarded with no entry written to $error.

Finally, let’s try the Stop preference:

PS> Get-Item -Path c:\, nosuchfile, c:\test, c:\nosuchfolder `

-ErrorAction Stop

Directory:

Mode LastWriteTime Length Name

---- ------------- ------ ----

d--hs- 02/05/2016 09:51 C:\

Get-Item : Cannot find path 'C:\test\nosuchfile' because it does not exist.

At line:1 char:1

+ Get-Item -Path c:\, nosuchfile, c:\test, c:\nosuchfolder -ErrorAction ...

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nosuchfile:String) [Get-Item], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetItemCommand

This time, you see only one output message and one error message—the first one. This is because the error is treated as a terminating error and execution stops.

#### Note

In earlier versions of PowerShell, the error message contained additional text explaining that execution stopped because of the error action preference setting. This is no longer the case in PowerShell v5.

The -ErrorAction parameter controls the error behavior for exactly one cmdlet. If you want to change the behavior for an entire script or even a whole session, you can do so by setting the $ErrorActionPreference variable. Let’s redo the last example but use the variable instead of the parameter:

PS> & {

$ErrorActionPreference = 'Stop'

Get-Item -Path c:\, nosuchfile, c:\test, c:\nosuchfolder

}

Directory:

Mode LastWriteTime Length Name

---- ------------- ------ ----

d--hs- 02/05/2016 09:51 C:\

Get-Item : Cannot find path 'C:\test\nosuchfile' because it does not exist.

At line:3 char:1

+ Get-Item -Path c:\, nosuchfile, c:\test, c:\nosuchfolder

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound:

(C:\test\nosuchfile:String) [Get-Item], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetItemCommand

Again, the cmdlet stops at the first error instead of continuing.

#### Note

In this example, note the use of the call operator & with a scriptblock containing the scope for the preference setting. Using the pattern & { …script text… }, you can execute fragments of script code so that any variables set in the enclosed script text are discarded at the end of the scriptblock. Because setting $ErrorActionPreference has such a profound effect on the execution of the script, we’re using this technique to isolate the preference setting.

Through the -ErrorActionPreference parameter and the $ErrorActionPreference variable, the script author has good control over when errors are written and when they’re terminating. Nonterminating errors can be displayed or discarded at will. But what about terminating errors? How does the script author deal with them? Sometimes you want an error to terminate only part of an operation. For example, you might have a script move a set of files using a series of steps for each move. If one of the steps fails, you want the overall move operation to terminate for that file, but you want to continue processing the rest of the files. To do this, you need a way to manage these terminating errors or exceptions, and that’s what we’ll discuss next.

## 14.2 Dealing with Errors That Terminate Execution

This section will deal with the ways that PowerShell processes errors that terminate the current flow of execution, also called terminating errors. If you have a programming background, you’re probably more familiar with terminating errors when they’re called by their more conventional name—exceptions. Call them what you will; we’re going to delve into catching these terminating errors. We’ll look at ways to trap or catch these errors and take action as a consequence. In some cases, these may be remedial actions (such as trying to fix the problem) or recording that the errors occurred.

The only way for exceptions to be caught in PowerShell v1 was by using the trap statement, which is somewhat similar to the on error statement in Visual Basic or VBScript. A better approach is the try/catch statement, modeled after the try/catch statement in C#, which was introduced in PowerShell v2.

#### The trap statement

Accepted best practice within the PowerShell community is to use try/catch rather than trap. We’ll cover the trap statement for completeness.

The trap statement can appear anywhere in a block of code. This means that it may be specified after a statement that generates an error and still handle that error. When an exception (terminating error) occurs that isn’t otherwise handled, control will be transferred to the body of the trap statement, and the statements in the body are then executed.

You can optionally specify the type of exception to catch, such as division by zero. But this can cause issues because traps don’t unwrap the underlying error, so they always see an ActionPreferenceStopException with originally nonterminating errors emitted from cmdlets. If no exception is specified, then it will trap all exceptions.

PS>trap { "Got it!" } 1/$null

Got it!

Attempted to divide by zero.

At line:1 char:21

+ trap { "Got it!" } 1/$null

+ ~~~~~~~

+ CategoryInfo : NotSpecified:(:) [], RuntimeException

+ FullyQualifiedErrorId : RuntimeException

What happens after a trap handler execution has completed depends on how the block finishes. If the body of the trap handler block finishes normally, an error object will be written to the error stream, and, depending on the setting of $ErrorActionPreference, either the exception will be rethrown or execution will continue at the statement after the statement that caused the exception.

You can control the interpreter’s behavior after you leave the trap handler by the break and continue keywords. Exiting a trap block using break is somewhat equivalent to the error action preference SilentlyContinue. Using continue ensures that the exception is handled and the error doesn’t bubble on.

The exception that was trapped is available in the trap block in the $\_ variable.

PS>$zero = 0

PS>trap { "Got it:$\_";continue } 1/$zero;

Got it:Attempted to divide by zero.

After the trap statement has completed, control transfers to the next statement in the same scope as the trap statement.

### 14.2.1 The try/catch/finally Statement

The trap statement, although powerful and flexible, ended up being hard to use for many of the traditional script/programming error-handling patterns. To address this, PowerShell v2 introduced the more familiar try/catch/finally statement found in other languages. As is the case with all of the other PowerShell flow-control statements, this statement adopts the syntax from C#.

There are three parts to this statement: the try block, the catch block, and the finally block. The try block is always required along with at least one of the catch or finally clauses. If an error occurs in the code in the try block resulting in an exception, PowerShell checks to see if there is a catch block specified. If there is a catch block, then it checks to see if specific exception types are to be caught. If at least one of the specified types matches, then the catch block is executed. If not, then the search continues looking for another catch block that might match.

#### Note

This is one place where the PowerShell try/catch statement has some advantages over its C# cousin. In C#, only one exception can be specified per catch clause, so it’s more complicated to take the same action for multiple exceptions that don’t have a common base class.

If there’s a catch block with no exception types specified, this clause will be executed (which tends to be the most common case). And if there’s a finally block, the code in the finally block runs. (The finally block always runs whether or not there was an exception.) Here’s an example using a catch statement with no exception type specified:

PS> try {

1

2

3/$null

4

5

}

catch {

"ERROR: $\_"

}

finally {

'ALL DONE'

}

1

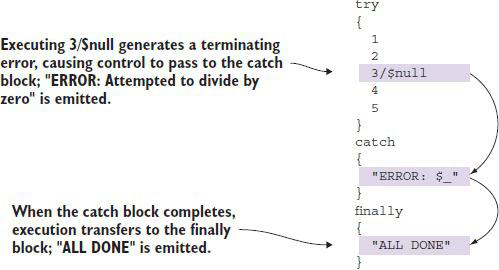
2

ERROR: Attempted to divide by zero.

ALL DONE

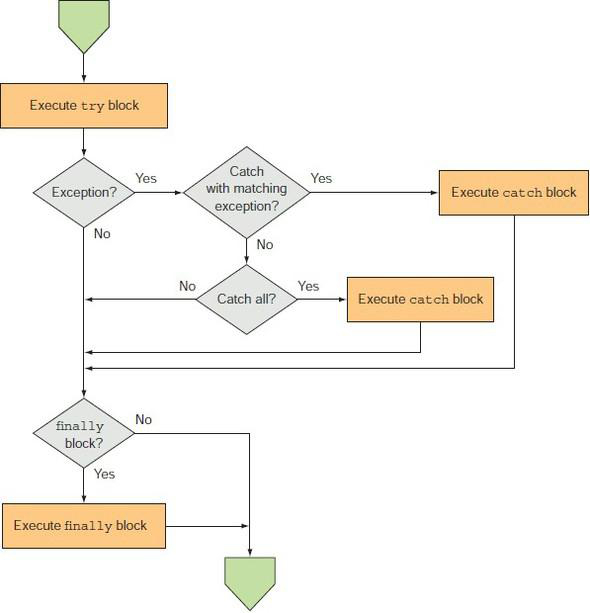
In this example, the third statement in the try block causes a terminating error. This error is caught and control transfers to the catch block. Then, when the catch block is complete, the finally block is executed. This flow of control is shown in [figure 14.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-142-115.xhtml#ch14fig05).

Figure 14.5: The flow of control in a try/catch/finally statement. When an exception occurs, control transfers to the catch block and then the finally block



The complete processing logic for the try/catch/finally statement is shown in the flowchart in [figure 14.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-142-115.xhtml#ch14fig06).

Figure 14.6: The complete logical flow in the try/catch/finally statement



#### Using try/catch in Expressions

An interesting application of the try/catch statement when used in combination with PowerShell’s expression-oriented syntax is that it makes it fairly easy to write functions that provide default values if an expression throws an exception. Let’s look at using try/catch in a custom div function. We want a function that never throws an exception even when dividing by zero. The function might look like this:

function div {

param

(

[int]$x,

[int]$y

)

try {

$x/$y

}

catch {

[int]::MaxValue

}

}

Give it a try

PS> div 1 0

2147483647

and you get the maximum integer value instead of the error you normally get when you divide by zero.

Clearly, exceptions are a powerful error-handling mechanism. With this mechanism, errors are never missed because you forgot to check for a return code. In fact, you have to do the opposite and take action to suppress them instead. Having mastered catching other people’s exceptions, let’s look at how you can leverage this feature in your own scripts with the throw statement.

### 14.2.2 The throw Statement

To complete the exception-handling topic, you need a way to generate terminating errors or exceptions. You can accomplish this by using the throw statement.

#### Note

In the original design, throw was supposed to be a cmdlet rather than a keyword in the language. But having a cmdlet throw the exception meant that the thrown exception was subject to the cmdlet’s error action policy, and the whole point of throw was to bypass this policy and always generate an exception. It wasn’t so much a case of the tail wagging the dog as it was staple-gunning the poor beast to the floor. And so, learning from past mistakes, Microsoft made it into a keyword.

The simplest example is to throw nothing:

PS> throw

ScriptHalted

At line:1 char:1

+ throw

+ ~~~~~

+ CategoryInfo :

OperationStopped: (:) [], RuntimeException

+ FullyQualifiedErrorId : ScriptHalted

This approach is convenient for casual scripting. You don’t need to create an error object or exception object—the throw statement takes care of all of that. Unfortunately, the message you get isn’t too informative. If you want to include a meaningful message, you can easily provide your own:

PS> throw 'My Message!'

My Message!

At line:1 char:1

+ throw 'My Message!'

+ ~~~~~~~~~~~~~~~~~~~

+ CategoryInfo :

OperationStopped: (My Message!:String) [], RuntimeException

+ FullyQualifiedErrorId : My Message!

You see the message in the output. It’s also possible to use throw to throw Error-Record objects or .NET exceptions if you want to use more detailed error handling. Instead of passing a string, you pass these objects.

To complete this chapter, we’ll look at the data available in the event log. The event log is the central store for log messages from the system as well as from all the applications, services, and drivers running on that machine. It’s a one-stop shop for diagnostic information. You’ll see how to access this diagnostic treasure trove using PowerShell.

## 14.3 PowerShell and the Event Log

The Windows event log provides a central place where applications and operating system components can record events like the starting and stopping of an operation, progress, and system and application errors. For system administration, having access to the event log is critical. As an admin tool, PowerShell support for the event log is quite important, so that’s what we’re going to look at in this section.

### 14.3.1 The EventLog Cmdlets

PowerShell v1 had only a single, fairly limited command (Get-EventLog) for working with the event log. More sophisticated operations required using the underlying .NET classes. PowerShell v2 filled this gap and now there’s a comprehensive set of cmdlets for working with the event log, as shown in [table 14.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-143-116.xhtml#ch14table03).

#### Show-EventLog

You may be wondering why PowerShell includes this cmdlet—all it does is launch the event log viewer. The answer is simple: usability. PowerShell is a command-line shell, so you should be able to launch GUI applications from the command line. You can, but there’s a small problem:most of the commands you want to run, including GUI commands, have names that aren’t obvious. For example, to launch the control panel applet for adding and removing software, you run appwiz.cpl. To change the display settings, run desk.cpl. These command names, though related to their function, are certainly not obvious to a user. Similarly, the command to start the event viewer is eventvwr.msc. In contrast, the Show-EventLog cmdlet, which follows the PowerShell naming guidelines, can easily be intuited once you know the rules. The next question is why provide a cmdlet instead of an alias? Because, as well as command naming, a cmdlet provides standard parameter handling, which allows for things like tab completion. By providing a “shim” cmdlet for the existing application, one more small bump is removed from the command-line user’s experience.

| **Table 14.3: The PowerShell EventLog cmdlets** | | |
| --- | --- | --- |
| **Cmdlet name** | **PowerShell version** | **Description** |
| Get-EventLog | v1, enhanced in v2 | Gets the events in an event log, or a list of the event logs, on the local or remote computers |
| Clear-EventLog | v2 | Deletes all entries from specified event logs on the local or remote computers |
| Write-EventLog | v2 | Writes a new event log entry to the specified event log on the local or remote computer |
| Limit-EventLog | v2 | Sets the event log properties that limit the size of the event log and the age of its entries |
| Show-EventLog | v2 | Displays the event logs of the local or a remote computer using the event viewer MMC console |
| New-EventLog | v2 | Creates a new event log and a new event source on a local or remote computer |
| Remove-EventLog | v2 | Deletes an event log or unregisters an event source |

The Get-EventLog cmdlet is what we’ll focus our attention on here. This cmdlet allows you to retrieve a list of the available application and system event logs and then look at the content of each of the logs. To get a list of the available logs, run Get-EventLog -List. The output will look something like this:

PS> Get-EventLog -List

Max(K) Retain OverflowAction Entries Log

------ ------ -------------- ------- ---

20,480 0 OverwriteAsNeeded 1,607 Application

20,480 0 OverwriteAsNeeded 0 HardwareEvents

512 7 OverwriteOlder 0 Internet Explorer

20,480 0 OverwriteAsNeeded 0 Key Management Service

128 0 OverwriteAsNeeded 9 OAlerts

20,480 0 OverwriteAsNeeded 3,171 Security

20,480 0 OverwriteAsNeeded 1,927 System

15,360 0 OverwriteAsNeeded 1,569 Windows PowerShell

In addition to the names of the logs, you can see the configuration settings for each log, such as the amount of space the log might take and what happens when the log fills up. You can use the Limit-EventLog cmdlet to change these limits for a log:

PS> Limit-EventLog -LogName Application -MaximumSize 25mb

then verify that the limit has been changed:

PS> Get-EventLog -List | where Log -match 'Application'

Max(K) Retain OverflowAction Entries Log

------ ------ -------------- ------- ---

25,600 0 OverwriteAsNeeded 1,607 Application

As well as listing the available logs, Get-EventLog lets you see the events in any log. Because the event logs can be quite large, the cmdlet supports a variety of options to control the amount of data returned. [Table 14.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-143-116.xhtml#ch14table04) describes the various Get-EventLog filter parameters.

| **Table 14.4: The types of filters provided by the Get-EventLog cmdlet** | |
| --- | --- |
| **Filter** | **Description** |
| Source | The -Source parameter allows you to filter log entries based on the name used to register the event source. This name is usually the name of the application logging the events, but for larger applications, it may be the name of a subcomponent within that application. |
| Message | The -Message parameter allows the retrieved entries to be filtered based on the event’s message text. The specified filter strings may contain wildcard patterns. (Note that because the text of a message is usually translated, the use of the -Message filter may not be portable to different locations.) |
| InstanceID | The InstanceId for an entry is the message resource identifier for the event. This identifier is used to retrieve the localized text for a message from the resource file for the registered event source. Because this identifier isn’t localized, the -InstanceID parameter provides a way to filter events by message that’s portable across locales because the message text is localized but the resource ID is always the same value. |
| EntryType | The entry type (or severity level) is a way of classifying events based on the potential impact of the corresponding event on the system’s behavior. The entry types are Information, Warning, Error, and Critical. Two additional event types can occur in the security log: Success Audit and Failure Audit. |
| UserName | The -UserName parameter filters based on the name of the user on whose behalf the event occurred. Wildcards patterns can be used in arguments to this parameter. |
| Before | Gets only the events that occur before the specified date and time. |
| After | Gets only the events that occur after the specified date and time. |

Let’s see how these parameters are used by working through a few examples. We’ll look at the Application log.

Start by listing the newest 10 events in this log:

PS> Get-EventLog -LogName Application -Newest 10

Index Time EntryType Source InstanceID Message

----- ---- --------- ------ ---------- -------

34931 May 09 20:39 Warning Outlook 1073741851 OAB Downlo...

34930 May 09 20:36 Information Outlook 1073741851 Starting O...

34929 May 09 20:32 Information NVWMI 1090519043 runPipeSer...

34928 May 09 20:32 Information NVWMI 1090519043 runPipeSer...

34927 May 09 20:32 Information NVWMI 1090519043 runPipeSer...

34926 May 09 20:32 Information NVWMI 1090519043 NVWMI - Ba...

34925 May 09 20:32 Information NVWMI 1090519043 runPipeSer...

34924 May 09 20:32 Information NVWMI 1090519043 runPipeSer...

34923 May 09 20:32 Information NVWMI 1090519043 NVWMI - Ba...

34922 May 09 20:32 Information NVWMI 1090519043 runPipeSer...

The -Index parameter lets you retrieve a specific entry from the log. Use Format-List to display additional properties of the entry:

PS> Get-EventLog -LogName Application -Index 34931 | Format-List

Index : 34931

EntryType : Warning

InstanceId : 1073741851

Message : OAB Download Failed. (Result code in event data).

Category : (0)

CategoryNumber : 0

ReplacementStrings : {OAB Download Failed. (Result code in event data).}

Source : Outlook

TimeGenerated : 09/05/2017 20:39:01

TimeWritten : 09/05/2017 20:39:01

UserName :

Now retrieve events using this message’s InstanceID:

PS> Get-EventLog -LogName Application -InstanceId 1073741851 -Newest 5

Index Time EntryType Source InstanceID Message

----- ---- --------- ------ ---------- -------

34932 May 09 20:41 Information Outlook 1073741851 Starting O...

34931 May 09 20:39 Warning Outlook 1073741851 OAB Downlo...

34930 May 09 20:36 Information Outlook 1073741851 Starting O...

34906 May 09 20:32 Warning Outlook 1073741851 OAB Downlo...

34904 May 09 20:31 Information Outlook 1073741851 Starting O...

You can use -Before and -After to retrieve messages around a specific date (and time if desired):

PS> Get-EventLog -LogName Application -After 'April 30/2017' `

-Before 'May 3/2017'

Here you retrieve all the messages between May 1 and May 2 in 2017. You can combine -Before, -Newest, -Message, and -After to perform further filtering. For example, to retrieve the last 10 messages on May 2, use this:

PS> Get-EventLog -LogName Application -Before 'May 3/2017' -Newest 10

or, you can use -Message and -After to find all messages matching a specific pattern that occurred after a specific date. For this example, use the month and day numbers and let the year default to the current year:

PS> Get-EventLog -LogName Application -Message '\*Defender\*' `

-After 'April 30/2017' |

Format-List UserName, TimeGenerated, EntryType, Message

You’ll see a number of records of this form:

UserName :

TimeGenerated : 09/05/2017 13:00:07

EntryType : Information

Message : Updated Windows Defender status successfully to

SECURITY\_PRODUCT\_STATE\_ON.

Why is all this useful? Imagine you see a critical error in an application. This error shows up in the Application log. You suspect that it might be related to either a hardware issue or a bad device driver. Rather than manually poring over hundreds of log entries, you can use the date from the Application log entry to retrieve the events in the System log that occurred shortly before the application.

Digging through the entries, you identify the problem that led to the failure. From this, you get the Source and InstanceID identifying the problematic entry. You quickly write a script to remediate the problem on this machine but realize that there may be other machines in the organization with similar issues. You put together a list of potentially at-risk machines and pass this list to Get-EventLog using the -ComputerName parameter. You also specify the -Source and -InstanceID parameters of the problematic message. This command will search the event logs of all the at-risk machines, returning a list of event log entries matching the criteria. From this set of events, you can get the names of all the computers that need to be fixed. Finally, you can use PowerShell remoting to run the remediation script on all the machines with the problem.

#### Note

Although you need PowerShell remoting to run the remediation script on the target machines, PowerShell remoting isn’t used when you use Get-EventLog to access a remote computer. Get-EventLog uses its own remoting protocol. This means you can use Get-EventLog to examine the logs of the target computer to help diagnose what went wrong using its own built-in remoting to connect to that computer. It’s not dependent on PowerShell remoting.

The Get-EventLog filtering capabilities make this kind of forensic analysis easy. One of the things you might want to analyze is PowerShell itself.

### 14.3.2 Examining the PowerShell Event Log

When PowerShell is installed, the installation process creates a new event log called Windows PowerShell. As PowerShell executes, it writes a variety of information to this log, which you can see using the Get-EventLog cmdlet. Let’s use the cmdlet to get the last few records from the PowerShell event log. As always, you can use the tools PowerShell provides to filter and scope the data you want to look at. You’ll use an array slice to get the last five records from the log:

PS> Get-EventLog -LogName 'Windows PowerShell' | select -Last 5

Index Time EntryType Source InstanceID Message

----- ---- --------- ------ ---------- -------

5 Nov 03 11:15 Information PowerShell 600 Provider "Fi...

4 Nov 03 11:15 Information PowerShell 600 Provider "Fu...

3 Nov 03 11:15 Information PowerShell 600 Provider "En...

2 Nov 03 11:15 Information PowerShell 600 Provider "Al...

1 Nov 03 11:15 Information PowerShell 600 Provider "Re...

The default presentation of the event records doesn’t show much information. Let’s look at one event in detail and see what it contains:

PS> Get-EventLog -LogName 'Windows PowerShell' |

select -Last 1 | Format-List \*

First, you get some basic event log elements common to all event log entries:

EventID : 600

MachineName : brucepayquad

Data : {}

Index : 1

Next, you see the event category. This isn’t the same as the error category discussed earlier. PowerShell event log entries are grouped into several large categories:

Category : Provider Lifecycle

CategoryNumber : 6

Next is the entry type and a message describing the entry. This is followed by a collection of detail elements, which includes things such as the state transition for the engine as well as some of the versioning information you saw on the $host object earlier. This is included in case you have multiple hosts for a particular engine:

EntryType : Information

Message : Provider "Registry" is Started.

Details:

ProviderName=Registry

NewProviderState=Started

SequenceNumber=1

HostName=ConsoleHost

HostVersion=5.1.14393.0

HostId=ee0ff0ec-0be8-49ab-8c47-beed57a906e7

HostApplication=C:\Windows\System32\

WindowsPowerShell\v1.0\powershell.exe

EngineVersion=

RunspaceId=

PipelineId=

CommandName=

CommandType=

ScriptName=

CommandPath=

CommandLine=

Source : PowerShell

The following fields specify the replacement strings that are available. These strings are substituted into the log message text:

ReplacementStrings : {Registry, Started, ProviderName=Registry

NewProviderState=Started

SequenceNumber=1

HostName=ConsoleHost

HostVersion=5.1.14393.0

HostId=ee0ff0ec-0be8-49ab-8c47-beed57a906e7

HostApplication=C:\Windows\System32\

WindowsPowerShell\v1.0\powershell.exe

EngineVersion=

RunspaceId=

PipelineId=

CommandName=

CommandType=

ScriptName=

CommandPath=

CommandLine=}

Finally, you get additional information for identifying the event log entry and when it occurred:

InstanceId : 600

TimeGenerated : 03/11/2016 11:15:13

TimeWritten : 03/11/2016 11:15:13

UserName :

Site :

Container :

Granted, the output isn’t all that interesting, but when you’re trying to figure out what went wrong on your systems, being able to see when the PowerShell interpreter was started or stopped could be useful. There are also certain types of internal errors (also known as bugs) that may cause a PowerShell session to terminate. These errors also will be logged in the PowerShell event log.

So far, we’ve looked at the classic event logs that have always been available in Windows. A new type of event log was introduced with Windows Vista; unfortunately, Get-EventLog and the other cmdlets listed in [table 14.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-143-116.xhtml#ch14table03) don’t work with these logs. You have to use Get-WinEvent.

### 14.3.3 Get-WinEvent

When working with the new Windows Event Log technology in Windows Vista and later, you have only Get-WinEvent. There are no cmdlets that can perform the other tasks listed in [table 14.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-143-116.xhtml#ch14table03).

#### Note

Get-WinEvent can also read classic event log backup files (.evt files), so it can be used to analyze any files of that type you may need to access.

In this section, we’ll show the differences in the way Get-EventLog and Get-WinEvent filter data from the event logs. We’ll start by examining the full list of available event logs:

PS> Get-WinEvent -ListLog \*

You’ll see the classic event logs Get-EventLog displays and then a large number of logs, most of them named Microsoft-Windows-<something>. A standard Windows 10 machine has approximately 400 of these logs. You can view a subset of the available logs:

PS> Get-WinEvent -ListLog Microsoft-Windows-PowerShell\* |

select Logmode, RecordCount, LogName

LogMode RecordCount LogName

------- ----------- -------

Circular 0 Microsoft-Windows-PowerShell-Des...

Retain 0 Microsoft-Windows-PowerShell/Admin

Circular 358 Microsoft-Windows-PowerShell/Operational

Get-WinEvent can filter event log data using a hashtable or an XPath query. We’ll use the hashtable syntax to re-create the filtering examples we used in [section 14.3.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-143-116.xhtml#ch14lev2sec7). The first task is to select the newest five entries in the log:

PS> Get-WinEvent -LogName Application | select -First 5

ProviderName: Outlook

TimeCreated Id LevelDisplayName Message

----------- -- ---------------- -------

09/05/2017 20:58:00 27 Information Starting OAB dow...

09/05/2017 20:52:51 27 Warning OAB Download Fai...

09/05/2017 20:41:43 27 Information Starting OAB dow...

09/05/2017 20:39:01 27 Warning OAB Download Fai...

09/05/2017 20:36:47 27 Information Starting OAB dow...

You can select single records by the RecordId property:

PS> Get-WinEvent -LogName Application | where RecordId -eq 34935

ProviderName: Outlook

TimeCreated Id LevelDisplayName Message

----------- -- ---------------- -------

09/05/2017 20:52:51 27 Warning OAB Download Fai...

Unfortunately, RecordId isn’t part of the default output, so you’ll have to format your display to include that data if you require access to it.

You can search the event log for events that occur in a specific time period:

PS> $start = (Get-Date).AddDays(-2)

PS> $end = (Get-Date).AddDays(-1)

PS> Get-WinEvent -FilterHashtable @{LogName='Application';

StartTime=$start; EndTime=$end}

You’ll notice that the ProviderName is supplied as part of the output. The records are displayed in chronological order—youngest first unless the -Oldest parameter is used to reverse the order. It may be advantageous to view the records with all the records from a single provider grouped together, with the individual records in descending time order:

PS> Get-WinEvent -FilterHashtable @{LogName='Application';

StartTime=$start; EndTime=$end} |

Sort-Object -Property @{Expression='ProviderName';

Descending=$false},

@{Expression='TimeCreated';Descending=$true}

Sort-Object will sort the records obtained by Get-WinEvent so that the provider names are in alphabetical order and the records for each provider are displayed from youngest to oldest. You can view the records from a single provider:

PS> Get-WinEvent -FilterHashtable @{Logname='Application'; ProviderName='SecurityCenter'; StartTime=$start; EndTime=$end}

That’s all we’re going to cover on event logs in this chapter. From these examples, you can see that the event logs provide a lot of information, much of which can help you manage and maintain your systems. The trick is being able to extract and correlate the information across the various logs, and this is where PowerShell can be very useful.

## 14.4 Summary

* There are two types of errors in PowerShell: terminating and nonterminating.
* Error records are written directly to default output.
* Error records are rich objects.
* The $error variable stores the last 256 errors (by default).
* You can specify a specific variable for errors by using the -ErrorVariable parameter.
* $? stores a Boolean value indicating execution status of the last command.
* $LASTEXITCODE stores the exit code of the last console command or exit statement but isn’t affected by cmdlets or .NET code.
* $ErrorActionPreference and the -ErrorAction parameter can be used to control the action taken if an error occurs.
* Terminating errors and exceptions can be managed by the trap statement or the try/catch/finally statements (preferred).
* Use the throw statement to generate your own terminating exceptions.
* The Get-EventLog cmdlet reads classic event logs.
* Get-WinEvent must be used for the new style event logs.

All these error features are great for letting you know something is wrong, but how do you go about fixing the problem? That’s the topic of the next chapter.

## Chapter 15: Debugging

### Overview

This chapter covers

* Creating script instrumentation
* Capturing session output
* The PowerShell debugger
* Command-line debugging
* Debugging PowerShell jobs, runspaces, and remote scripts

*Big Julie: “I had the numbers taken off for luck, but I remember where the spots formerly were.”  
Guys and Dolls, words and music by Frank Loesser*

No one writes code that always works correctly the first time it’s run. When the worst happens and your code won’t run, or deliver the correct results, you need to debug it to find the problem or problems. Start by adding statements that track your code’s execution and capture session output and then move on to more advanced techniques using the PowerShell debugger on running code. The techniques we’ll show you in this chapter will enable you to find and fix code problems much faster.

Let’s start by looking at how you can provide instrumentation for your scripts so they provide you with diagnostic information.

## 15.1 Script Instrumentation

The most basic form of debugging a script is to put statements in it, using the Write\* cmdlets, that display information about the script’s execution. The Write\* cmdlets separate your debugging output from the rest of the output by displaying the debugging output directly on the console.

#### Note

These statements will slow down your code execution because they’ll still be parsed even if they aren’t run. You may need to remove them in production code if execution speed is of paramount importance.

Checking the code before it runs can catch a number of potential errors as well as help you implement best practices in your coding.

### 15.1.1 The Write\* Cmdlets

A number of cmdlets enable you to write out information during the execution of your scripts:

PS> Get-Command write\* -Module Microsoft.PowerShell.Utility |

Format-Wide -Column 3

Write-Debug Write-Error Write-Host

Write-Information Write-Output Write-Progress

Write-Verbose Write-Warning

#### Output streams

The Write\* cmdlets are closely tied to the output streams of the same name. The Error, Warning, Verbose, and Progress streams are targeted at the end user of the script. Information targets the operator; Debug targets the developer.

Error is self-explanatory with the complication of terminating and nonterminating errors. It tracks what went wrong.

Warning is for things that might be wrong; for example, not including an Import-DSCResource statement for the default resources is a warning. It’s used infrequently because most conditions should be treated as errors. The PSDesiredStateConfiguration.psm1 module has a number of Write-Warning statements.

Verbose is for giving the user more detailed information about the behavior of the operation they requested (example: Copy-Item -Verbose). It tracks what’s happening in detail.

Debug is used by developers to instrument code to make it easier to discover and analyze bugs in their scripts. Unlike Write-Host, Debug statements can be added to a script and left in place to assist in the debugging process later on or in the field (with, as mentioned, the caveat that they add execution overhead). As an example, the PSDesiredStateConfiguration.psm1 module has lots of Debug statements. Debug tracks information that’s useful for figuring out why the script misbehaved and for locating bugs.

Information is a new stream in PowerShell v5 and is targeted at the operations team rather than the immediate user. It should be used to track operational behavior. For example, in DSC, it would be used to track what state checks are being done and consequently why an operation is being performed. For instance, in DSC, a file was missing so a new file will be created. It’s the equivalent of writing to the analytic log.

Progress tracks simple progress as a percentage. In many cases, using Verbose is more useful than Progress, but people like progress bars. A progress bar will slow down execution of your code by a significant amount.

Knowing what’s tracked in each stream enables you to target your debugging efforts correctly.

You’ve seen Write-Output in action throughout the book. It does what it says and outputs whatever it’s passed to the next step in the pipeline—or if it’s at the end of pipeline, to the default output mechanism, usually the screen:

PS> 1..3 | foreach {$psitem | Write-Output}

1

2

3

You don’t need to use this cmdlet because the default action at the end of a pipeline is to display objects on the pipeline. It’s useful when you want to force output.

Write-Progress isn’t considered debugging as such, but it will display a progress bar during execution of one or more commands. Tracking the progress of your code’s execution may supply clues if something goes wrong. Run the following in the console and the ISE and observe the results:

$max = 10000

1..$max |foreach {

Write-Progress -Activity Test -PercentComplete (($psitem/$max)\*100)

}

Interestingly the code runs much quicker in the ISE.

#### Note

Outputting progress activity will slow down execution of your code.

The Write-Host cmdlet is the way that most people start creating script instrumentation:

PS> 1..3 | foreach {

$x = $psitem \* 2

Write-Host -Object "$psitem doubled is: $x"

}

1 doubled is: 2

2 doubled is: 4

3 doubled is: 6

The drawback is that your output from Write-Host is mixed in with your code output:

PS> 1..3 | foreach {

$x = $psitem \* 2

Write-Host -Object "$psitem doubled is: $x"

$y += $x

$y

}

1 doubled is: 2

2

2 doubled is: 4

6

3 doubled is: 6

12

Also, using Write-Host is an all-or-nothing proposition—you can’t turn it on and off to suit your needs. Ideally, you want to separate the output of your instrumentation, at least visually, and control whether the instrumentation is active. You can use the -ForegroundColor and -BackgroundColor parameters to control the text colors for Write-Host, but there are better approaches.

#### Verbose and Debug

Write-Verbose and Write-Debug enable you to output information from your script, or function, when you need it. You can turn off the information during normal usage and turn it on when you have a problem. Debug is for outputting developer debug information when your code runs, similar to the debug log in the event log. The Verbose stream is for giving the end user more information about the operation they requested.

#### Note

The Information stream is for information about how the operation is proceeding, equivalent to the analytic log in the event log.

Today in DSC, verbose messages are recorded in the analytic log. If the Information stream had been around when DSC was started, it would have been using Information and not Verbose.

You use these two cmdlets together with the [CmdletBinding()] attribute that we discussed in [chapter 7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-7-59.xhtml#ch07). Consider this simple function:

function fdvtest {

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$true)]

[int]$i,

[int]$mult=2

)

PROCESS {

$i \* $mult

}

}

The output is

PS> 1..3 | fdvtest

2

4

6

Look at the function’s syntax:

PS> Get-Command fdvtest -Syntax

fdvtest [[-i] <int>] [[-mult] <int>] [<CommonParameters>]

The CommonParameters include -Verbose and -Debug. Before you can use those parameters and get any sensible output, you need to add the appropriate statements. Let’s say you want to see a message before the calculation. You can use Write-Verbose:

function fdvtest {

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$true)]

[int]$i,

[int]$mult=2

)

PROCESS {

Write-Verbose -Message 'Performing multiplication'

$i \* $mult

}

}

Running the function without the -Verbose switch gives you the same output as previously. When you run the function with the -Verbose switch

PS> 1..3 | fdvtest -Verbose

VERBOSE: Performing multiplication

2

VERBOSE: Performing multiplication

4

VERBOSE: Performing multiplication

6

you get a clearly labeled message that serves to separate your debugging messages from the normal output. The verbose messages are in a different color (defaults are yellow in the console and cyan in the ISE) than the normal output to give further emphasis. Write-Verbose can output messages that enable you to track the progress of your code.

You can control the color for verbose, debug, warning, error, and information messages using Tools > Options in the ISE and then selecting Output Streams on the Colors and Fonts tab. In the console, you can view the colors for the streams using the following:

PS> $host.PrivateData

ErrorForegroundColor : Red

ErrorBackgroundColor : Black

WarningForegroundColor : Yellow

WarningBackgroundColor : Black

DebugForegroundColor : Yellow

DebugBackgroundColor : Black

VerboseForegroundColor : Yellow

VerboseBackgroundColor : Black

ProgressForegroundColor : Yellow

ProgressBackgroundColor : DarkCyan

The colors can be modified. If you don’t like red for the error color:

PS> $host.PrivateData.ErrorForegroundColor = 'Green'

Your errors are now shown in green! The console settings are on a session basis, so you need to put the changes into your profile if you want them to be applied to all your console sessions.

Debug messages can also be added to your code:

function fdvtest {

[CmdletBinding()]

param(

[Parameter(ValueFromPipeline=$true)]

[int]$i,

[int]$mult=2

)

BEGIN {

Write-Debug "`$mult = $mult"

}

PROCESS {

Write-Verbose -Message 'Performing multiplication'

Write-Debug -Message "`$i = $i"

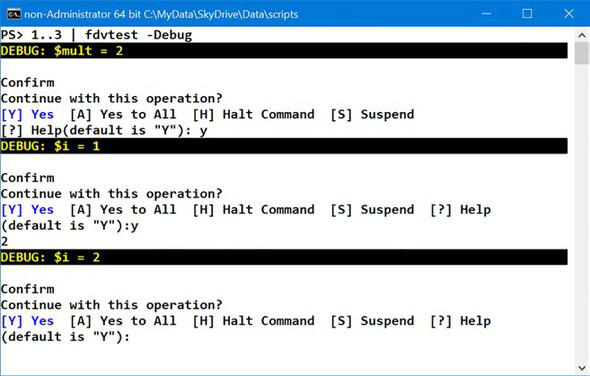
$i \* $mult

}

}

In this case, a debug message has been added to give the value of the multiplier and the value of $i. When you run the function with the -Debug parameter, you’ll see a dialog each time the code reaches a Write-Debug statement, as shown in [figure 15.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-151-119.xhtml#ch15fig01).

Figure 15.1: Output when using the -Debug functionality



If you halt or suspend the command, you can step into it and use the standard PowerShell debugging functionality, which we’ll get to soon.

#### Note

When you use the -Verbose or -Debug switch, it will turn on the appropriate output for all cmdlets in your script or function as well as enabling the appropriate Write cmdlet.

#### Error, Warning, and Information

In [chapter 4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-4-32.xhtml#ch04) you saw how to redirect output to the Error, Warning, and Information streams. You can achieve the same thing by using the appropriate Write cmdlet. Unlike the Write-Verbose and Write-Debug cmdlets from the previous section, you don’t get a switch to enable the functionality at the script or function level.

When you use Write-Information or Write-Warning, further processing depends on the value of the appropriate preference variable (or parameter), as shown in [table 15.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-151-119.xhtml#ch15table01).

| **Table 15.1: Cmdlet and preference variable relationships** | | | |
| --- | --- | --- | --- |
| **Category** | **Cmdlet** | **Preference variable** | **Parameter** |
| Information | Write-Information | $InformationPreference | InformationAction |
| Warning | Write-Warning | $WarningPreference | WarningAction |
| Error | Write-Error | $ErrorActionPreference | ErrorAction |

You can view the current values of the preference variables like this:

PS> Get-Item variable:\*preference

Name Value

---- -----

ConfirmPreference High

DebugPreference SilentlyContinue

ErrorActionPreference Continue

ProgressPreference Continue

VerbosePreference SilentlyContinue

WarningPreference Continue

InformationPreference SilentlyContinue

WhatIfPreference False

Write-Error declares a nonterminating error, which won’t stop further processing. As you saw in [chapter 14](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-14-113.xhtml#ch14), you can use Throw to declare a terminating error.

#### Note

Starting in Windows PowerShell v5, Write-Host is a wrapper for Write-Information. It uses the Information stream instead of writing directly to the host. You can now use Write-Host to emit output to the Information stream, but the $InformationPreference preference variable and InformationAction common parameters don’t affect Write-Host messages.

As usual, the easiest way to explain the use of these cmdlets is with an example:

1..7 |

foreach {

switch ($psitem) {

1 {

Write-Information -MessageData "Starting. Value is $\_" `

-InformationAction Continue

Break

}

5 {

Write-Warning -Message "Nearly Finished. Value is $\_"

Break

}

7 {

Write-Error -Message "Value of $\_ is too high" `

-ErrorAction Continue

}

default {

Write-Information -MessageData "Value is $\_" `

-InformationAction Continue

}

}

}

When you run this code, you’ll see this output:

Starting. Value is 1

Value is 2

Value is 3

Value is 4

WARNING: Nearly Finished. Value is 5

Value is 6

: <truncated for brevity as repeat of code>

Value of 7 is too high

+ CategoryInfo : NotSpecified: (:) [Write-Error], WriteErrorException

+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException

Notice that the output from Write-Warning has a WARNING: prefix to draw your attention, but that from Write-Information does not. It is in the standard text color.

When you looked at the event logs in [chapter 14](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-14-113.xhtml#ch14), you saw that events were logged under the categories of Information, Warning, and Error. If you prefer to have the messages from your code instrumentation written to the event log, you’ll use Write-EventLog. You can even create your own specific event log, as you’ll see next.

### 15.1.2 Writing Events to the Event Log

You can write your Information, Warning, and Error messages to the console, but that assumes that you’ll be watching execution of the script. If you’re running the code in the middle of the night, this isn’t an optimum solution. Using the event logs enables you to capture the messages for future analysis. It’s possible to use one of the existing event logs, but you can create your own:

PS> New-EventLog -LogName PiALog -Source Scripts

You need to provide a name for the log and an event source. Administrative privileges are required to create event logs. You can have multiple sources per log file, but source names must be unique on the machine—you can’t use the same source name for sources writing to two different logs. You can create additional sources:

PS> New-EventLog -LogName PiALog -Source Functions

You can view the sources associated with an individual log file like this:

PS> Get-CimInstance -ClassName Win32\_NTEventLogFile `

-Filter "LogFileName='PiALog'" |

select -ExpandProperty Sources

PiALog

Functions

Scripts

Now that you have your log file, let’s modify the code from the example in the previous section to use your new log:

1..7 |

foreach {

switch ($psitem) {

1 {

Write-EventLog -Message "Starting. Value is $\_" `

-LogName PiALog -Source Scripts -EntryType Information `

-EventId 1001

Break

}

5 {

Write-EventLog -Message "Nearly Finished. Value is $\_" `

-LogName PiALog -Source Scripts -EntryType Warning `

-EventId 1010

Break

}

7 {

Write-EventLog -Message "Value of $\_ is too high" `

-LogName PiALog -Source Scripts -EntryType Error `

-EventId 1020

}

default {

Write-EventLog -Message "Value is $\_" `

-LogName PiALog -Source Scripts -EntryType Information `

-EventId 1002

}

}

}

When you run this code, you won’t see any output. You can view the records written into the event log like so:

PS> Get-EventLog -LogName PiALog

Index Time EntryType Source InstanceID Message

----- ---- --------- ------ ---------- -------

7 May 10 11:47 Error Scripts 1020 Value of 7 is too high

6 May 10 11:47 Information Scripts 1002 Value is 6

5 May 10 11:47 Warning Scripts 1010 Nearly Finished. Va...

4 May 10 11:47 Information Scripts 1002 Value is 4

3 May 10 11:47 Information Scripts 1002 Value is 3

2 May 10 11:47 Information Scripts 1002 Value is 2

1 May 10 11:47 Information Scripts 1001 Starting. Value is 1

Using the features described in this section and the previous section, you can instrument your scripts in order to debug their behavior. Although this is a tried-and-true way of debugging, it’s reactive, and you can’t work with the script while it’s running. PowerShell provides other mechanisms to find problems in your scripts. One of these features is strict mode, our next topic.

### 15.1.3 Catching Errors with Strict Mode

PowerShell provides built-in static and runtime checks to help you catch errors in your scripts. Static checks are performed at script load/compile time, and runtime checks are dynamic checks done at runtime.

#### Note

These features are similar to Option Explicit in Visual Basic or strict mode in PERL and are named after the PERL feature.

PowerShell v1 could check for undefined variables through the Set-PSDebug cmdlet. PowerShell v2 introduced Set-StrictMode, which enables a much more comprehensive set of checks. Set-StrictMode affects only the current scope and any child scopes, so you can use it in scripts without affecting the global scope and therefore other scripts.

#### Note

Strict mode turns on certain checks in regular scripts and functions. PowerShell classes (see [chapter 19](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-19-143.xhtml#ch19)) have their own built-in set of strict checks that are always enabled. The checks in classes are intended to support more robust programming, as opposed to scripting, in PowerShell. If you’re writing large applications in PowerShell, using classes can result in more robust code because of these additional checks.

You can control the checks that are performed by using the -Version parameter, which takes 1, 2, or Latest as an argument controlling whether v1 or v2 checks are enabled, as shown in [table 15.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-151-119.xhtml#ch15table02).

| **Table 15.2: Strict mode versions** | |
| --- | --- |
| **Version** | **Action** |
| 1 | Prohibits references to uninitialized variables, except for uninitialized variables in strings |
| 2 | As 1, plus:   * Prohibits references to uninitialized variables (including uninitialized variables in strings) * Prohibits references to nonexistent properties of an object * Prohibits function calls that use the syntax for calling methods * Prohibits a variable without a name (${}) (PowerShell v5 checks for this with strict mode turned off) |
| Latest | Selects the latest (strictest) version available |

Unless you have a specific need to do otherwise, it’s usually recommended to use Latest as your version.

#### Catching Uninitialized Variable Use in String Expansions

Strict mode in v1 caught only references to uninitialized variables in script text. It didn’t catch the use of uninitialized variables in string expansions. Strict mode v2 fixes this, and the use of uninitialized variables is caught everywhere. In non-strict mode, for example, when you reference a nonexistent variable, it’s treated as being equivalent to $null.

Now turn on strict mode v1 and reference a nonexistent variable:

PS> Set-StrictMode -Version 1

PS> $nosuchvariable

The variable '$nosuchvariable' cannot be retrieved because it

has not been set.

At line:1 char:1

+ $nosuchvariable

+ ~~~~~~~~~~~~~~~

+ CategoryInfo :

InvalidOperation: (nosuchvariable:String) [], RuntimeException

+ FullyQualifiedErrorId : VariableIsUndefined

You get the uninitialized variable message as expected. Now put the string in quotes

PS> "$nosuchvariable"

and it expands the string with no errors. Turn on strict mode v2 and try the string expansion:

PS> Set-StrictMode -Version 2

PS> "$nosuchvariable"

The variable '$nosuchvariable' cannot be retrieved because it

has not been set.

At line:1 char:2

+ "$nosuchvariable"

+ ~~~~~~~~~~~~~~~

+ CategoryInfo :

InvalidOperation: (nosuchvariable:String) [], RuntimeException

+ FullyQualifiedErrorId : VariableIsUndefined

You also get the uninitialized variable error in the string expansion case.

#### Catching Attempts to Read Nonexistent Properties

To have appropriately shell-like behavior, by default PowerShell allows you to try dereferencing nonexistent properties. That means you can do things like display a mixed collection of [System.IO.FileInfo] and [System.IO.DirectoryInfo] objects, including a reference to the Length property that doesn’t exist for [System.IO.DirectoryInfo] objects. Imagine how annoying it would be to type dir and get a lot of “property not found” errors. Try running

PS> dir | foreach { $\_.name + " " + $\_.length }

in your home directory with strict mode v2 turned on, and you’ll see what we mean.

#### Note

This applies only to explicit property references in script text. Cmdlets still ignore missing properties even when strict mode v2 is turned on. The interactive environment is pretty much unusable otherwise.

Try a simple example. First, turn off strict mode and then get a [DateTime] object into the variable $date:

PS> Set-StrictMode -Off

PS> $date = Get-Date

Now reference a nonexistent property

PS> $date.nosuchproperty

and no error is raised. Now turn on strict mode and try accessing the property:

PS> Set-StrictMode -Version Latest

PS> $date.nosuchproperty

The property 'nosuchproperty' cannot be found on this object.

Verify that the property exists.

At line:1 char:1

+ $date.nosuchproperty

+ ~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : NotSpecified: (:) [], PropertyNotFoundException

+ FullyQualifiedErrorId : PropertyNotFoundStrict

This time you get an error. As with the variable check, property checks will help catch typos in your script. It would be better if you could catch them at compile time, but then you’d need to know the types of all the expressions. Because PowerShell is dynamically typed, that isn’t possible.

#### Note

But what about checking against the type constraints on variables, you might ask? Strict mode could include this kind of check, but it can’t do a complete check because PowerShell allows extensions on instances as well as types. For example, when you look at a file entry in PowerShell, you see a Mode property. The underlying .NET type [System.IO.FileInfo] doesn’t have a property with this name. The Mode property is one of the properties added by the PowerShell runtime. Because these properties can be added at runtime, even for a type-constrained variable, the most you could say is that the member probably won’t exist by the time the statement is executed. Only the runtime check is guaranteed to be correct.

#### Checking for Functions Called like Methods

An extremely common source of errors for experienced programmers is to call functions in the same way you would in other languages or in the same way methods are called in PowerShell. Let’s see this in action. Turn off strict mode:

PS> Set-StrictMode -Off

then define a function that looks like this:

PS> function divide ($x,$y) { $x / $y }

This function takes two arguments, divides the first by the second, and returns the result. Now let’s call it like a method, with parentheses around a function. This is how you’d call a function in a language like C#:

PS> divide(9, 3)

Method invocation failed because [System.Object[]]

does not contain a method named 'op\_Division'.

At line:1 char:27

+ function divide ($x,$y) { $x / $y }

+ ~~~~~~~

+ CategoryInfo : InvalidOperation:

(op\_Division:String) [], RuntimeException

+ FullyQualifiedErrorId : MethodNotFound

What happens is that you get a surprising error. You know that numbers can be divided, so why does this fail? By putting the two arguments in parentheses, you’re telling the system to pass a single argument, which is an array of two numbers. We talked about this problem in [section 6.2.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-62-53.xhtml#ch06lev2sec3). Now turn on strict mode and try it again:

PS> Set-StrictMode -Version Latest

PS> divide(9, 3)

The function or command was called as if it were a method. Parameters should

be separated by spaces. For information about parameters, see the about\_

Parameters Help topic.

At line:1 char:1

+ divide(9, 3)

+ ~~~~~~~~~~~~

+ CategoryInfo : InvalidOperation: (:) [], RuntimeException

+ FullyQualifiedErrorId : StrictModeFunctionCallWithParens

This time you get a prescriptive error message explaining exactly what’s gone wrong. Follow the instructions, rewriting the function call, removing the parameters, and separating it with spaces instead of a comma, and then try running it again:

PS> divide 9 3

3

This time it works.

This technique may seem like a trivial, almost silly check, but this issue has caused many problems for many people, including members of the PowerShell team.

#### Applying Strict Mode to Scripts

You now know what the checks are—let’s talk about when to apply them. In general, it’s recommended that new code be written to be strict mode Latest “clean.” The code should produce no errors when strict mode Latest is turned on. The temptation is to leave it on all the time.

Unfortunately, this approach can break a lot of script code. Many scripts are written to take advantage of the default property dereference behavior. That means a lot of fixing may be necessary. There are also cases where rewriting the code to not depend on this behavior can be messy—the code would have to either explicitly check for the existence of a property before trying to access it or explicitly trap the exception and ignore it.

#### Note

Our recommendation is to use strict mode when developing but ensure it’s turned off in production.

Consider the example at the beginning of the section that addressed catching references to nonexistent properties:

PS> dir | foreach { $\_.name + " " + $\_.length }

This code results in an error every time dir returns a directory object. To make this work in strict mode Latest, you’d have to do something like

PS> dir | foreach { $\_.name + " " + $(try { $\_.length } catch { $null })}

where the try/catch statement is used to process the error. In this code, if there’s no exception, then the value of the property is returned. If there is an exception, the catch block returns $null. (At least the expression-oriented nature of the PowerShell language simplifies this example instead of requiring intermediate variables and an if statement.)

### 15.1.4 Static Analysis of Scripts

Most of the checks performed in strict mode are applied only at runtime, but there are some other checks you can do statically before you ever run the script. This was made possible in PowerShell v2 by the introduction of the PowerShell tokenizer API, a .NET class that takes the text of a PowerShell script and breaks it down into pieces called tokens.

Tokens correspond to the types of elements found in the PowerShell language, which include things like keywords and operators—all the things we talked about in [chapters 2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-2-16.xhtml#ch02) through [8](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-8-66.xhtml#ch08). Unfortunately, this mechanism isn’t packaged in a convenient way for scripting. It was designed for the PowerShell ISE, but with a little work it’s still usable from a script. First, we’ll discuss how to use the API. We’ll start by tokenizing a small piece of script text. If you have strict mode turned on, you’ll have to turn it off for these examples:

PS> Set-StrictMode -Off

Put the text you want to tokenize into a variable:

PS> $script = "function abc ($x) {dir; $x + 1}"

The tokenizer returns two things: the tokens that make up the script and a collection of any errors encountered while parsing the script. Because the API is designed for use from languages that can’t return multiple values, you also need to create a variable to hold these errors:

PS> $parse\_errs = $null

Now you’re ready to tokenize the script. Do so by calling the static method Tokenize() on the PSParser class as follows:

PS> $tokens = [System.Management.Automation.PSParser]::

Tokenize($script,[ref] $parse\_errs)

This code will put the list of tokens in the $tokens variable, and any parse errors will be placed into a collection in $parse\_errs. Now dump these two variables—$parse\_errs to the error stream and $tokens to the output stream:

PS> $parse\_errs | Write-Error

PS> $tokens | Format-Table -AutoSize Type,Content,StartLine,StartColumn

Type Content StartLine StartColumn

---- ------- --------- -----------

Keyword function 1 1

CommandArgument abc 1 10

GroupStart ( 1 14

GroupEnd ) 1 15

GroupStart { 1 17

Command dir 1 18

StatementSeparator ; 1 21

Operator + 1 24

Number 1 1 26

GroupEnd } 1 27

Because the text being tokenized is a valid PowerShell script, no errors are generated. You do get a list of all the tokens in the text displayed on the screen. You can see that each token includes the type of the token, the content or text that makes up the token, as well as the start line and column number of the token. You’ll now wrap this code into a function to make it easier to call. Name the function Test-Script:

function Test-Script {

param (

[Object]$script

)

$parse\_errs = $null

$tokens = [system.management.automation.psparser]::

Tokenize($script,[ref] $parse\_errs)

$parse\_errs | Write-Error

$tokens

}

Try it on a chunk of invalid script text:

PS> Test-Script "function ($x) {$x + }" |

Format-Table -AutoSize Type,Content,StartLine, StartColumn

Test-Script : System.Management.Automation.PSParseError

At line:1 char:1

+ Test-Script "function ($x) {$x + }" |

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : NotSpecified: (:) [Write-Error], WriteErrorException

+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,Test-Script

Test-Script : System.Management.Automation.PSParseError

At line:1 char:1

+ Test-Script "function ($x) {$x + }" |

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : NotSpecified: (:) [Write-Error], WriteErrorException

+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,Test-Script

Test-Script : System.Management.Automation.PSParseError

At line:1 char:1

+ Test-Script "function ($x) {$x + }" |

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : NotSpecified: (:) [Write-Error], WriteErrorException

+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,Test-Script

Test-Script : System.Management.Automation.PSParseError

At line:1 char:1

+ Test-Script "function ($x) {$x + }" |

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : NotSpecified: (:) [Write-Error], WriteErrorException

+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,Test-Script

Type Content StartLine StartColumn

---- ------- --------- -----------

Keyword function 1 1

GroupStart ( 1 10

GroupEnd ) 1 11

GroupStart { 1 13

Operator + 1 15

GroupEnd } 1 17

Now you see a number of errors. When you run a script that has syntax errors, you get one error before the parsing continues. With the tokenizer API, the parser tries to reset itself and continue. This means that you may be able to deal with more errors at one time, but the reset process doesn’t always work and sometimes you get incorrect errors.

#### Note

In many cases the first error is the culprit, and correcting that also removes subsequent errors in the report. But it’s not always the case, so be prepared for several passes through the process.

The other thing to notice is that in the list of tokens being displayed, some of the tokens in the script, such as the variables, aren’t output. Again, this is because when the parser attempts to recover, it can get confused and miss some tokens. That’s why when you run a script you get only one error displayed. You know the first error displayed by the tokenizer is correct but aren’t sure about the rest. It’s simpler, if not more efficient, to deal with one correct error at a time rather than a collection of possible incorrect errors.

Let’s rewrite the test function. You’re going to do a little work to clean up the errors, but you’ll also add a new static check. Because the tokenizer output tells you what tokens are commands, you can use Get-Command to see if there are any references to commands that don’t exist. This won’t always be an error—a script may load a module defining the missing command at runtime—so you need to consider it a warning to investigate instead of an error. Here’s what the new script looks like:

function Test-Script {

param (

[Object]$script

)

$parse\_errs = $null

$tokens = [system.management.automation.psparser]::

Tokenize($script, [ref] $parse\_errs)

foreach ($err in $parse\_errs)

{

'ERROR on line ' +

$err.Token.StartLine +

': ' + $err.Message +

"`n"

}

foreach ($token in $tokens)

{

if ($token.Type -eq 'CommandArgument')

{

$gcmerr = Get-Command $token.Content 2>&1

if (! $? )

{

'WARNING on line ' +

$gcmerr.InvocationInfo.ScriptLineNumber +

': ' + $gcmerr.Exception.Message +

"`n"

}

}

}

}

The first part of the script hasn’t changed much—you tokenize the string and then display any errors, though in a more compact form. Then you loop through all of the tokens looking for code commands. If you find a command, you check to see if it exists. If not, you display a warning.

Let’s try it out. First, define the test script with expected errors and an undefined command:

$badScript = @'

for ($a1 in nosuchcommand)

{

while ( )

$a2\*3

}

'@

Now run the test and see what you get:

PS> Test-Script $badScript

ERROR on line 1: Unexpected token 'in' in expression or statement.

ERROR on line 3: Missing expression after 'while' in loop.

ERROR on line 3: Missing statement body in while loop.

WARNING on line 17: The term 'nosuchcommand' 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.

In the output you see the expected syntax errors, but you also get a warning for the undefined command. You could do many things to improve this checker, such as looking for variables that are used only once. By using these analysis techniques on the script text, you can find potential problems much sooner than you would if you waited to hit them at runtime.

#### Note

We include this section to introduce the basic concept of tokens and the tokenizer. In practice, there are powerful static analysis tools that do far more than what we’ve looked at in these simple examples. The PSScriptAnalyzer, created by the PowerShell team, provides an extensive set of rules for analyzing your code. There’s also a third-party commercial offering tool called ISESteroids, which provides an excellent interactive code analysis experience inside the PowerShell ISE. Both of these tools are available from the PowerShell Gallery. Using these tools is highly recommended.

So far, we’ve looked at a number of tools and approaches that you can use to learn what’s wrong with your scripts. But how do you figure out what’s going on when other people are running your (or other people’s) scripts in a different environment, possibly at a remote location? To help with this, PowerShell includes a session transcript mechanism. You’ll learn how this works in the next section.

## 15.2 Capturing Session Output

When trying to debug what’s wrong with someone’s script at a remote location, you’ll find it extremely helpful to see the output and execution traces from a script run. PowerShell allows you to do this via a mechanism that captures console output in transcript files. This transcript capability is exposed through the Start-Transcript and Stop-Transcript cmdlets.

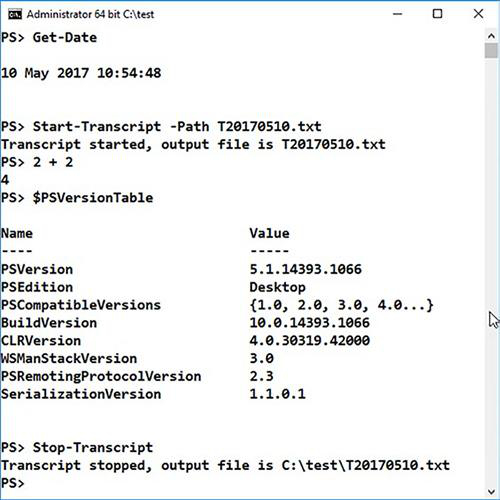
#### Note

Up until PowerShell v5, the implementation of these cmdlets is a feature of the console host (PowerShell.exe) and so is not available in other hosts, including the PowerShell ISE. This changed in PowerShell v5 when the transcript functionality was added to the PowerShell ISE. Other host applications may have similar mechanisms.

### 15.2.1 Starting the Transcript

To start a transcript, run Start-Transcript, as shown in the next example. Let’s begin by running a command before starting the transcript so you can see what is and is not recorded, as shown in [figure 15.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-152-120.xhtml#ch15fig02).

Figure 15.2: Using the transcript cmdlets



Run Get-Date to get the current date and then start the transcript. If you didn’t specify a filename for the transcript file, one will be automatically generated for you in your Documents directory. Now run a couple of additional commands and stop the transcript. Again, it conveniently tells you the name of the file containing the transcript:

Now let’s see what was captured:

PS> Get-Content -Path C:\test\T20170510.txt

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Windows PowerShell transcript start

Start time: 20170510105524

Username: W510W16\Richard

RunAs User: W510W16\Richard

Machine: W510W16 (Microsoft Windows NT 10.0.14393.0)

Host Application: powershell

Process ID: 8732

PSVersion: 5.1.14393.1066

PSEdition: Desktop

PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.14393.1066

BuildVersion: 10.0.14393.1066

CLRVersion: 4.0.30319.42000

WSManStackVersion: 3.0

PSRemotingProtocolVersion: 2.3

SerializationVersion: 1.1.0.1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Transcript started, output file is T20170510.txt

PS>

PS>2 + 2

4

PS>

PS>$PSVersionTable

Name Value

---- -----

PSVersion 5.1.14393.1066

PSEdition Desktop

PSCompatibleVersions {1.0, 2.0, 3.0, 4.0...}

BuildVersion 10.0.14393.1066

CLRVersion 4.0.30319.42000

WSManStackVersion 3.0

PSRemotingProtocolVersion 2.3

SerializationVersion 1.1.0.1

PS>

PS>Stop-Transcript

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Windows PowerShell transcript end

End time: 20170510105547

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The transcript file includes a header showing you the start time, the name of the user running the script, and the name and OS information about the computer on which the command is run.

You see the filename yet again because it was written out after transcription was turned on and so is captured in the transcript.

After that, you see the output of the commands you ran (including Stop-Transcript) and finally a trailer showing the time the transcript stopped.

### 15.2.2 What Gets Captured in the Transcript

It seems obvious that everything should get captured in the transcript file, but that isn’t the case in the early versions of PowerShell. The transcript captured everything written through the host APIs. What didn’t get captured was anything that bypasses these APIs and writes directly to the console. This missing information is most significant when you’re running applications like ipconfig.exe. If these commands weren’t redirected within PowerShell, then their output went directly to the console and bypassed the host APIs. Instead of running

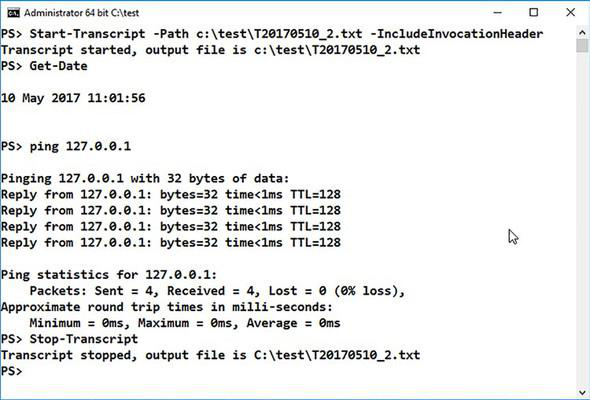
PS> cmd /c echo THIS WONT BE CAPTURED

you had to use

PS> cmd /c echo THIS WILL BE CAPTURED 2>&1 | Write-Host

New in Powershell v5 is the -IncludeInvocationHeader parameter, which adds a time stamp when commands are run. An example is shown in [figure 15.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-152-120.xhtml#ch15fig03).

Figure 15.3: Using the -IncludeInvocationHeader parameter in a transcript



The transcript file looks like this:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Windows PowerShell transcript start

Start time: 20170510110151

Username: W510W16\Richard

RunAs User: W510W16\Richard

Machine: W510W16 (Microsoft Windows NT 10.0.14393.0)

Host Application: powershell

Process ID: 8732

PSVersion: 5.1.14393.1066

PSEdition: Desktop

PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.14393.1066

BuildVersion: 10.0.14393.1066

CLRVersion: 4.0.30319.42000

WSManStackVersion: 3.0

PSRemotingProtocolVersion: 2.3

SerializationVersion: 1.1.0.1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Transcript started, output file is c:\test\T20170510\_2.txt

PS>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Command start time: 20170510110156

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PS>Get-Date

10 May 2017 11:01:56

PS>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Command start time: 20170510110204

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PS>ping 127.0.0.1

Pinging 127.0.0.1 with 32 bytes of data:

Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:

Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),

Approximate round trip times in milli-seconds:

Minimum = 0ms, Maximum = 0ms, Average = 0ms

PS>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Command start time: 20170510110218

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PS>Stop-Transcript

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Windows PowerShell transcript end

End time: 20170510110218

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Using the transcript cmdlets, it’s easy to have the remote user capture the output of their session. Have the remote user call Start-Transcript, run their script, and then call Stop-Transcript. This process will produce a transcript file that the user can send to you for examination.

So far, we’ve looked at ways of capturing information about code that’s executing. It’s time to go deeper and learn how to interactively debug your code to discover why things aren’t working as they should.

## 15.3 PowerShell Script Debugging Features

The PowerShell debugging tools have grown over the versions of PowerShell. PowerShell v1 didn’t include a debugger but did have some limited tracing capabilities. Version 2 introduced a much more comprehensive debugger along with graphical debugging support in the ISE. With PowerShell, you can now debug jobs, workflows, and PowerShell runspaces as well as processes.

The debugging tools available are outlined in [table 15.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15table03).

| **Table 15.3: Debugging tools** | | |
| --- | --- | --- |
| **PowerShell debugger** | **Debugger connectivity** | **Tracing** |
| Get-PSCallStack Disable-PSBreakpoint Enable-PSBreakpoint Get-PSBreakpoint Remove-PSBreakpoint Set-PSBreakpoint | Debug-Job Debug-Process Debug-Runspace Disable-RunspaceDebug Enable-RunspaceDebug Get-RunspaceDebug Set-PSDebug Wait-Debugger | Get-TraceSource Set-TraceSource Trace-Command |

We’ll start by looking at the limited (but still useful) tracing features carried over from v1. Then you’ll learn how to debug from the ISE. You’ll see the command-line debugger and the additional capabilities it has to offer, including debugging jobs, workflows, and remote scripts. Finally, we’ll look at the command-tracing capabilities.

### 15.3.1 The Set-PSDebug Cmdlet

The Set-PSDebug cmdlet can be used to set the PowerShell v1 strict mode, although as shown in [section 15.1.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-151-119.xhtml#ch15lev2sec3) Set-StrictMode gives more options. The debugger has subsumed what Set-PSDebug does, making the cmdlet effectively redundant. We’ll mention the remaining features for completeness but strongly advise you to use the debugger for these actions.

#### Tracing Statement Execution

You turn on basic script tracing as follows:

PS> Set-PSDebug -Trace 1

In this trace mode, each statement executed by the interpreter will be displayed on the console. The debugging output is prefixed with the DEBUG: tag and is typically shown in a different color than normal text. Note that the entire script line is displayed. This means that if you have a loop all on one line, you’ll see the line repeated.

#### Note

This is a good reason, even though PowerShell doesn’t require it, to write scripts with one statement per line: it can help with debugging, both when tracing and when using the debugger to set breakpoints.

Basic tracing doesn’t show you any function calls or scripts you’re executing. You don’t see when you enter the function. To get this extra information, you need to turn on full tracing:

PS> Set-PSDebug -Trace 2

DEBUG: 1+ >>>> Set-PSDebug -Trace 2

#### Note

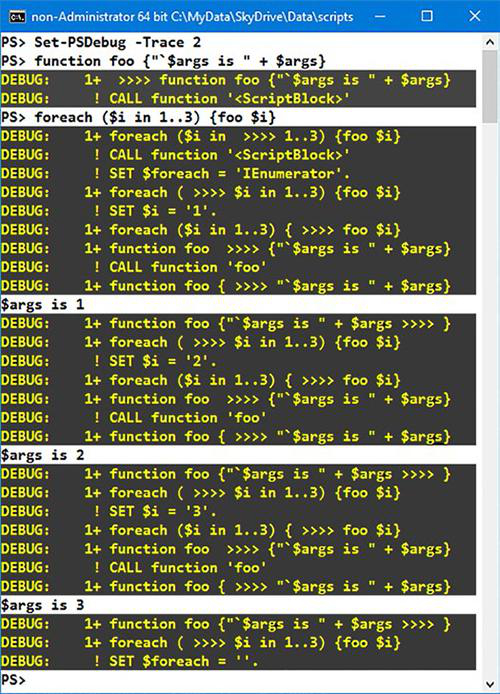
You only see the DEBUG:.. output if you already have tracing enabled. Use Set-PSDebug -Trace 0 to turn tracing off.

Now define a function:

PS> function foo {"`$args is " + $args}

When you execute the function in this mode, you also see the function calls, as shown in [figure 15.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15fig04).

Figure 15.4: Tracing function calls



In addition to function calls, full tracing adds to the display by showing variable assignments.

For each iteration in the loop, tracing shows the following:

* Loop iteration
* Function call
* Statement doing the assignment
* Assignment to $x, including the value assigned
* Statement that emits the value

The value displayed is the string representation of the object being assigned, truncated to fit in the display. It depends on the ToString() method defined for that object to decide what to display. For arrays and other collections, it shows you a truncated representation of the elements of the list. Overall, script tracing is pretty effective, but sometimes you still need to add calls to the Write cmdlets, as discussed in [section 15.1.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-151-119.xhtml#ch15lev2sec1).

#### Debugging scripts run by other people

The other thing to remember is PowerShell’s transcript capability. Transcripts combined with tracing provide a valuable tool to help with debugging scripts that are being run by other people in your organization. By capturing the trace output in a transcript file, you can get a much better idea of what a script is doing in the other user’s environment.

Tracing is also valuable in debugging remote scripts where you can’t use the ISE debugger, as you’ll see later in this chapter.

#### Stepping Through Statement Execution

The next debugging feature we’ll look at is the mechanism that PowerShell provides for stepping through a script.

#### Note

Like the tracing mechanism, this stepping feature is also a carryover from PowerShell v1. It’s largely subsumed by the PowerShell debugger, but there are some advanced scenarios, such as debugging dynamically generated code, where it’s still quite useful. If you use [ScriptBlock]::Create() to dynamically generate a scriptblock, you can’t set a breakpoint because you don’t have a line number in a file to use to set the breakpoint. More on this later.

You turn stepping on by calling the Set-PSDebug cmdlet with the -Step parameter:

PS> Set-PSDebug -Step

#### Note

Using -Step automatically sets a Trace level of 1.

Rerun the foreach loop and take a look at the prompt that’s displayed:

PS> foreach ($i in 1..3) {foo $i}

Continue with this operation?

1+ foreach ($i in >>>> 1..3) {foo $i}

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help

(default is "Y"):

The interpreter displays the line to be executed and then asks the user to select Yes, Yes to All, No, or No to All. The default is Yes.

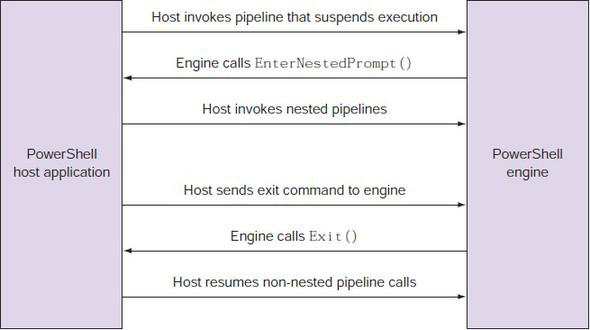
If you answer Yes, that line will be executed and you’ll be prompted as to whether you want to execute the next line. If you answer Yes to All, then step mode will be turned off and execution will continue normally. If you answer either No or No to All, the current execution will be stopped and you’ll be returned to the command prompt. There’s no difference in the behavior between No and No to All.

There’s one more option in the stepping list that we haven’t talked about: Suspend. This option is interesting enough to cover in its own section.

### 15.3.2 Nested Prompts and the Suspend Operation

One of the most interesting aspects of dynamic language environments is that a script can recursively call the interpreter. You saw this with the Invoke-Expression cmdlet in [chapter 10](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-10-82.xhtml#ch10). A variation is to recursively call the interpreter interactively. This means you are, in effect, suspending the currently running command and starting a new nested session. This sequence of events is illustrated in [figure 15.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15fig05).

Figure 15.5: Suspending execution and entering a nested prompt requires operations on both the host and engine sides of the session



In [figure 15.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15fig05), you see that the user makes a call to the engine using the interfaces provided by the host application. In this case, instead of returning to the caller, the engine calls back to the host indicating that it should enter a nested-prompt mode. While in nested-prompt mode, because the original command pipeline is still active (the engine never returned to the host), the host must now use nested pipelines to execute commands. This continues until the engine calls the Exit() API, usually in response to a request from the user, and the host can resume the original pipeline.

The net effect of all this is that you can suspend the currently executing PowerShell pipeline and interact with PowerShell at the nested prompt. Why is this interesting? Because it allows you to examine and modify the state of the suspended session by using the regular PowerShell commands you’re used to. Instead of creating a whole new language for debugger operations, you use the same language you’re debugging. This feature is the core of all of the debugging capabilities in PowerShell.

There are a couple of ways to enter a nested-prompt session, as you’ll see in the next two sections.

#### Suspending a Script While in Step Mode

The Suspend operation prompt shown during stepping creates a nested interactive session. Let’s try it. First, turn on stepping:

PS> Set-PSDebug -Step

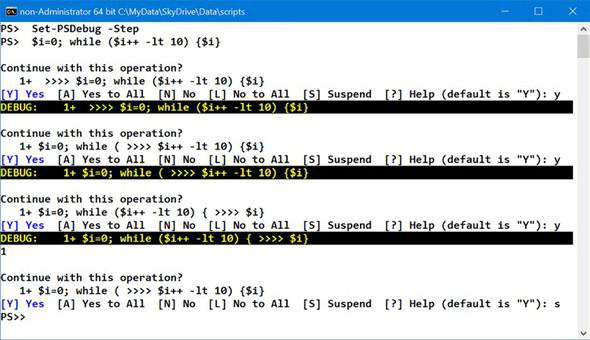
Then run a statement that should loop 10 times, printing out the numbers 1–10:

PS> $i=0; while ($i++ -lt 10) { $i }

You’ll see all the intermediate blather. Keep stepping until the first number is displayed.

At this point, use the Suspend operation to suspend stepping. When prompted, respond by typing s followed by pressing Enter instead of only pressing Enter. This leaves you at the position shown in [figure 15.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15fig06).

Figure 15.6: Suspending execution of a script



You immediately receive a new prompt. In [figure 15.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15fig06) the prompt changes from PS> to PS>>.

#### Note

The way to tell when you’re in nested-prompt mode is to check the $NestedPromptLevel variable. If you’re in a nested prompt, this variable will be greater than 0.

In this nested prompt, you can do anything you’d normally do in PowerShell. In this case, you want to inspect the state of the system. For example, let’s check to see what the variable $i is set to:

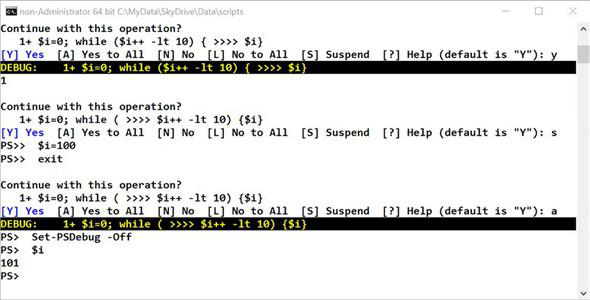
PS> $i

1

But you’re not limited to inspecting the state of the system: you can change it. Let’s make the loop end early by setting the value to something larger than the terminating condition. Set it to 100.

Now exit the nested-prompt session with the normal exit statement. This returns you to the previous level in the interpreter where, because you’re stepping, you’re prompted to continue. Respond by typing a followed by pressing Enter for [A] Yes to All to get out of step mode. You can turn off debugging and view the value of $i, as shown in [figure 15.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15fig07).

Figure 15.7: Modify the value of a variable while the script is suspended



There are two things to notice here: the loop terminates, and the value of $i is 101 (the loop incremented before it terminated).

#### Prompt

If you don’t see the change in prompt shown in [figures 15.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15fig06) and [15.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15fig07), check the prompt function in your profile. It should contain something like this:

$(if (Test-Path -Path variable:/PSDebugContext) {'[DBG]:'}

else {''}) + "PS$('>' \* ($nestedPromptLevel + 1)) "

This sets the prompt to PS> for normal use and adds a further > for each level of nesting that you enter. If you switch to the debugging prompt ([section 15.4.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-154-122.xhtml#ch15lev2sec9)), the prompt changes to [DBG]: PS>>. Another > character will be added if you enter a nested prompt while in debug mode.

You can see other options for including in your prompt in the about\_prompts help file.

Using the Suspend feature, you can stop a script at any point and examine or modify the state of the interpreter. You can even redefine functions in the middle of execution (although you can’t change the function that’s currently executing). This makes for a powerful debugging technique, but it can be annoying to use stepping all the time. Also, many users forget to end Suspend mode. This is where having a real debugger makes all the difference.

#### PowerShell debugger

With PowerShell v2, a powerful new debugger was added to the product. It can be used in graphical mode (in the ISE) or can be accessed through the debugging cmdlets. The operations of the graphical debugger are performed through the debugging cmdlets, so our discussion will center on them.

## 15.4 Command-Line Debugging

Given the nature of the PowerShell environment, you need to support debugging in a variety of environments. The most effective way to do that is to enable debugging scripts from the command line.

#### Note

The graphical debugger is built on top of the commands we’re going to cover in this section. Anything that can be done in the graphical debugger can be done from the command line, but the commands provide a great deal of power that isn’t exposed in the graphical debugger.

This makes it possible to use the debugger from the console host as well. As always, these debugging features are surfaced through a set of cmdlets. The cmdlets are listed in [table 15.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-154-122.xhtml#ch15table04).

| **Table 15.4: The PowerShell debugger cmdlets** | |
| --- | --- |
| **Cmdlet** | **Description** |
| Get-PSCallStack | Gets the current call stack |
| Enable-PSBreakPoint | Enables an existing breakpoint |
| Disable-PSBreakPoint | Disables a breakpoint without removing it |
| Set-PSBreakPoint | Sets a breakpoint |
| Get-PSBreakPoint | Gets the list of breakpoints |
| Remove-PSBreakPoint | Removes an existing breakpoint |

Command-line debugging is also important for another reason: There are many more things you can do using these cmdlets, including writing scripts to debug scripts. All the features you’ve seen in the GUI debugger are available from the command line, but not all the command-line features are available from the GUI. In fact, the GUI debugger surfaces only a portion of the functionality of what can be done with the PowerShell debugger. In the next few sections, we’ll dig into these capabilities.

### 15.4.1 Working with Breakpoint Objects

Let’s begin our discussion by looking at how breakpoints are implemented. So far, you’ve seen a fairly conventional debugger experience, but the introspective nature of PowerShell allows you to do much more when working with breakpoints. As with most everything else, breakpoints in PowerShell are objects (as you’ll see in a moment) that you can script against.

Breakpoints have an interesting property, -Action, which holds instances of our old friend, the scriptblock. By specifying actions in scriptblocks, breakpoints can do much more than interrupt execution when the breakpoint is hit. Using scriptblocks allows you to perform arbitrary actions controlling when or even whether the breakpoint fires. Let’s see how this works with a simple test script (save as testscript2.ps1):

PS> @'

"Starting"

$sum = 0

foreach ($i in 1..10)

{

$sum += $i

}

"The sum is $sum"

'@ > testscript2.ps1

This script loops over the numbers from 1–10, summing them and then printing the result. Now define a breakpoint for this script using the Set-PSBreakPoint command:

PS> $firstBP = Set-PSBreakpoint -Script testscript2.ps1 -Line 4 `

-Action {

if ($i -gt 3 -and $i -lt 7)

{

Write-Host "> DEBUG ACTION: i = $i, sum = $sum"

}

}

This command specifies that a scriptblock will be executed every time you hit line 4 in the test script. In the body of the scriptblock, you’re checking to see if the value of $i is greater than 3 and less than 7. If so, you’ll display a message. You have to use Write-Host to display this message because the results of the scriptblock aren’t displayed. The Set-PSBreakpoint command returns an instance of a breakpoint object. Let’s display it as a list so you can see its members:

PS> $firstBP | Format-List

Id : 1

Script : C:\test\testscript2.ps1

Line : 4

Column : 0

Enabled : True

HitCount : 0

Action :

if ($i -gt 3 -and $i -lt 7)

{

Write-Host "> DEBUG ACTION: i = $i, sum = $sum"

}

This code shows the full path to the script and the line in the script that will trigger the action as well as the action itself.

#### Note

The Id number may be different in your case depending on other actions you’ve taken in the console. Id numbers start at zero.

You can use Get-Member to examine the breakpoint object:

PS> Get-PSBreakpoint | Get-Member

TypeName: System.Management.Automation.LineBreakpoint

Name MemberType Definition

---- ---------- ----------

Equals Method bool Equals(System.Object obj)

GetHashCode Method int GetHashCode()

GetType Method type GetType()

ToString Method string ToString()

Action Property scriptblock Action {get;}

Column Property int Column {get;}

Enabled Property bool Enabled {get;}

HitCount Property int HitCount {get;}

Id Property int Id {get;}

Line Property int Line {get;}

Script Property string Script {get;}

In this output, you see some familiar bits of information: the breakpoint ID and the line and script where it applies. The HitCount property records the number of times a breakpoint has been hit. The Action property you’ve already met.

Run the test script to see how it works:

PS> .\testscript2.ps1

Starting

> DEBUG ACTION: i = 4, sum = 6

> DEBUG ACTION: i = 5, sum = 10

> DEBUG ACTION: i = 6, sum = 15

The sum is 55

The output shows the value of $i and $sum as long as $i is between 3 and 7 as intended.

Before we move on to the next example, remove all the breakpoints so they don’t confuse the results in the example:

PS> Get-PSBreakpoint | Remove-PSBreakpoint

#### Using the HitCount property

The HitCount property is interesting because the scriptblock can control whether or not a script breaks. You may want the script to break only after a certain number of iterations or if a variable has a specific value. Here’s an example:

Get-PSBreakpoint | Remove-PSBreakpoint

# Write the value each time

$null = Set-PSBreakpoint -line 19 -script example.ps1 -Action {

Write-Verbose -verbose "The value of num is $num"

}

# Break the fifth time the breakpoint has been encountered

$null = Set-PSBreakpoint -line 19 -script example.ps1 -Action {

if ($\_.HitCount -eq 5)

{

break

}

}

foreach ($num in 1 .. 10)

{

"Num is $num"

}

"I'm done"

The use of break and continue in breakpoint actions makes it possible to create arbitrary conditional breakpoints. This is powerful. Also, if you don’t break, then you can do things like trace execution in a fine-grained way. This is much more effective than Set-PSDebug -trace, for example.

This time, instead of only displaying a message, you’re going to use the break keyword to break the script under specific conditions. Here’s the command to define the new breakpoint:

PS> $firstBP = Set-PSBreakpoint -Script testscript2.ps1 -Line 4 `

-Action {

if ($i -eq 4)

{

Write-Host "> DEBUG ACTION: i = $i, sum = $sum"

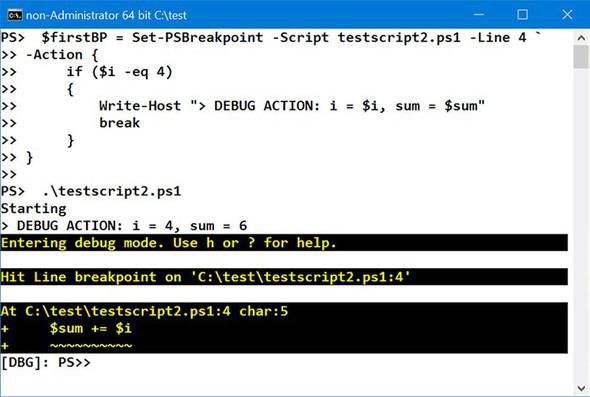
break

}

}

For this breakpoint, you’ll fire only the action on line 4 of the test script. In the scriptblock body, you’ll display the message as before and then call break, which will break the execution of the script, as shown in [figure 15.8](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-154-122.xhtml#ch15fig08).

Figure 15.8: Entering the command-line debugger



You have a number of options available at the break prompt. Type ? at the break prompt to view them. Use the c command to continue execution and complete the script. The completed script displays the sum. Don’t forget to clean up the breakpoint.

Now let’s move on to the next example.

### 15.4.2 Setting Breakpoints on Commands

The most common scenario using the debugger involves setting breakpoints on lines in a file, but it’s also possible to break on a specific command. Define a simple function

PS> function hello { 'Hello world!' }

and set a breakpoint on that function:

PS> Set-PSBreakpoint -Command hello

This time you won’t associate an action and you’ll allow the default behavior—causing a break in execution—to occur. Execute the function:

PS> hello

Hit Command breakpoint on 'hello'

At line:1 char:16

+ function hello { 'Hello world!' }

+ ~

[DBG]: PS>>

When the command is run, you immediately hit the breakpoint. Enter c and allow the function to complete. Among other things, the ability to set breakpoints on commands as opposed to specific lines in a script allows you to debug interactively entered functions.

Now let’s move on to the final example in this section: setting breakpoints on variables.

### 15.4.3 Setting Breakpoints on Variable Assignment

In the previous examples, the breakpoints were triggered when execution reached a certain line in the script or you entered a command. You can also cause a break when variables are read or written.

#### Note

You should always specify the script you’re debugging; otherwise, the breakpoint triggers whenever the variable changes with unpredictable and potentially undesired results.

In the following command, you’ll specify an action to take when the $sum variable is written:

PS> $thirdBP = Set-PSBreakpoint -Script testscript2.ps1 `

-Variable sum -Mode Write -Action {

if ($sum -gt 10)

{

Write-Host "> VARIABLE sum was set to $sum"

}

}

For this breakpoint, you’re using -Mode Write to specify that the breakpoint should trigger only when the variable is written. In practice, you could have omitted this because Write is the default mode (the other modes are Read and ReadWrite). Then in the action scriptblock, you’ll use Write-Host as before to display the value of $sum, but only when it’s greater than 10. Let’s see what this breakpoint looks like:

PS> $thirdBP | Format-List

Id : 2

Variable : sum

AccessMode : Write

Enabled : True

HitCount : 0

Action :

if ($sum -gt 10)

{

Write-Host "> VARIABLE sum was set to $sum"

}

You see the line, variable, and access mode that will trigger the action and the scriptblock to execute when triggered. Run the test script:

PS> .\testscript2.ps1

Starting

> VARIABLE sum was set to 15

> VARIABLE sum was set to 21

> VARIABLE sum was set to 28

> VARIABLE sum was set to 36

> VARIABLE sum was set to 45

> VARIABLE sum was set to 55

The sum is 55

You see the output messages from the action scriptblock. One of the nice things is that a variable-based breakpoint isn’t tied to a specific line number in the script, so it will continue to work even when you edit the script.

Although these examples are by no means exhaustive, they give you a sense of the capabilities of the PowerShell command-line debugger. You’re able to do much more sophisticated debugging from the command line. But even for the command line, there are a number of limitations to the debugging capabilities. We’ll look at these limitations in the final part of this section.

### 15.4.4 Debugger Limitations and Issues

The PowerShell debugger, though powerful, does suffer from a couple of limitations. The dynamic nature of the PowerShell language means that code can be created at any time and you aren’t always able to set breakpoints on this code. This is where the techniques you saw earlier in the chapter can help. You can insert the example breakpoint function into dynamic or anonymous code, allowing you to effectively set a breakpoint in that code.

Also, because variables are never declared, it’s not possible to specify an instance of a variable via its declaration; you can only select the target variable by name. Scoping a breakpoint to a particular file or command helps with correctly targeting the desired variable.

So far, we’ve looked at debugging PowerShell scripts on the local machine. Power-Shell v4 and v5 introduced the capability to debug other types of PowerShell commands and scripts running on remote machines.

## 15.5 Beyond Scripts

A number of cmdlets enable you to connect to PowerShell commands. They were listed in [table 15.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-153-121.xhtml#ch15table03) under the “Debugger connectivity” header. For convenience, they’re listed again here:

* Debug-Job
* Debug-Process
* Debug-Runspace
* Disable-RunspaceDebug
* Enable-RunspaceDebug
* Get-RunspaceDebug

This list shows that you now have the capability to debug jobs, processes, and PowerShell runspaces. In addition, you can debug workflows and scripts running on remote computers. Let’s see how these features work.

### 15.5.1 Debugging PowerShell Jobs

PowerShell jobs were introduced in [chapter 13](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-13-108.xhtml#ch13). A job runs in a separate PowerShell session that’s automatically created, used, and removed. Once the job starts, you don’t have any visibility inside the job. Now you can use the standard PowerShell debugging techniques to investigate your jobs.

Let’s run a simple (never–ending) job:

PS> $sb = {

$i = 0

while ($true) {

"My value is $i"

$i++

Start-Sleep -Seconds 5

}

}

PS> Start-Job -Name MyLongJob -ScriptBlock $sb

This sets a variable, $i, to 0. The code then loops through, listing and incrementing the variable with a five-second pause before running the next iteration of the loop. [Figure 15.12](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-155-123.xhtml#ch15fig12) shows this job running in the ISE.

The job starts and displays the standard job information as expected. You can use Debug-Job to open the job for debugging. A job can be accessed by Id, Name, InstanceId, or a job object. When you start debugging the job, you receive a message showing the line of code at which the job has stopped:

PS> Debug-Job -id 3

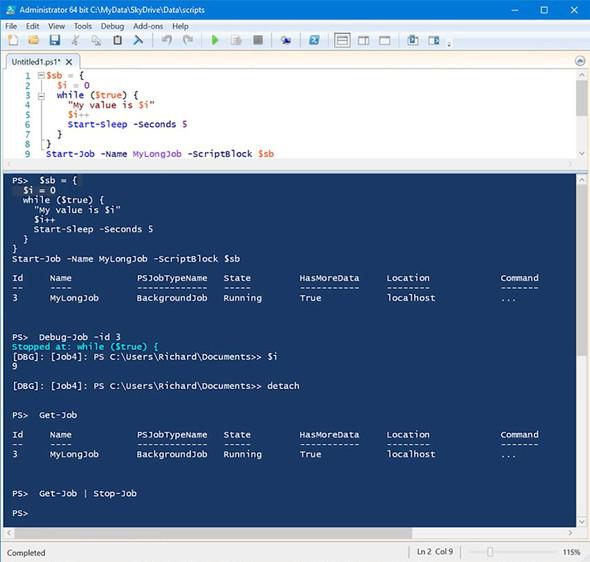
Stopped at: while ($true) {

[DBG]: [Job4]: PS C:\Users\Richard\Documents>> $i

9

The Debug-Job cmdlet attaches the ISE (or console) debugger to the job. You can then debug the script running in the job as if it were running interactively in the ISE (or the console). Once you’ve finished debugging, you can quit the debugger, by using the quit command, which will end debugging and stop the job or detach the debugger and allow the job to continue running, as shown in [figure 15.9](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-155-123.xhtml#ch15fig09).

Figure 15.9: Debugging a PowerShell job



Your prompt changes to the debug prompt, and you can apply standard debugging techniques such as investigating the value of variables. You can use the detach command to stop debugging and allow the job to continue.

#### Note

Don’t forget to stop the job and remove it!

This is a big step forward, but you don’t know where you’re going to enter the script that the job is running. In a similar manner to other PowerShell scripts, setting a breakpoint in the script the job is running gives you control of where debugging starts.

If you modify the previous example to be a script:

PS> @'

$i = 0

while ($true) {

"My value is $i"

$i++

Start-Sleep -Seconds 5

}

'@ > dbjob.ps1

you can set a breakpoint when you create the job:

PS> $job = Start-Job -ScriptBlock {

Set-PSBreakpoint -Script C:\test\dbjob.ps1 -Line 3

C:\test\dbjob.ps1

}

The job will run until it hits the breakpoint. It will then pause with a state of AtBreakpoint, as shown in [figure 15.10](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-155-123.xhtml#ch15fig10).

Figure 15.10: Using Set-PSBreakpoint in a PowerShell job



You’ll be presented with a debug prompt and can perform your debugging tasks. Use detach to leave debugging and return to normal job activity. PowerShell workflows that are running as jobs also can be debugged using the techniques in this section.

You may often find yourself running scripts on a remote machine. Debugging in this scenario has been problematic in the past but is now fully supported.

### 15.5.2 Debugging Remote Scripts

Debugging of remote scripts through the console was introduced in PowerShell v4. In PowerShell v5 this was extended to include debugging remote scripts through the ISE.

#### Note

The ISE (but not the console) can be used to open, edit, and save remote script files.

Put the dbjob.ps1 script from the previous section onto a remote machine. Remember that you can copy files to and from remote machines over PowerShell remoting sessions in PowerShell v5.

Create a remoting session to the machine on which you want to debug the script and enter the session:

PS> $s = New-PSSession -ComputerName server01

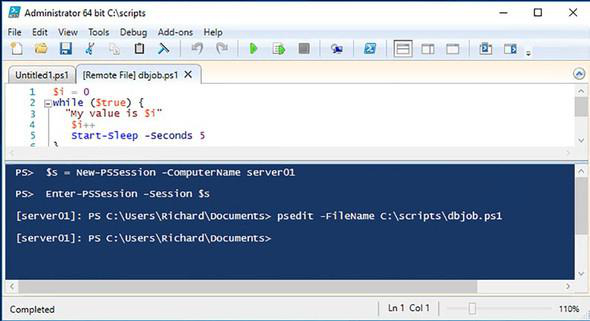
PS> Enter-PSSession -Session $s

You can then open the file for editing:

psedit -FileName C:\scripts\dbjob.ps1

[Figure 15.11](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-155-123.xhtml#ch15fig11) shows this in action.

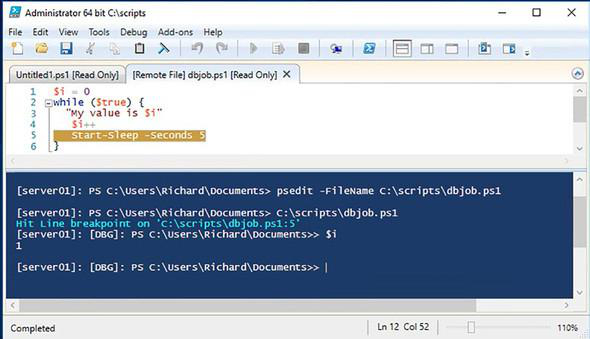
Figure 15.11: Editing and debugging a file on a remote server



Notice the [Remote File] decorator as part of the tab header for the file that’s being edited. When you edit a file in this manner, you’re accessing it across the remoting session; the file is copied across the session and stored locally in temporary storage. When changes to the file are saved, the file is copied back across the remoting session to the machine it came from.

You can set breakpoints in the file and the debugger will be opened when you run the file from the ISE, as shown in [figure 15.12](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-155-123.xhtml#ch15fig12).

Figure 15.12: Debugging a remote script



The debugger behaves in exactly the same way as when debugging a local script. When you’ve completed your debugging, type q to exit the debugger and stop script execution. The last debugging technique we want to show you is debugging PowerShell runspaces.

### 15.5.3 Debugging PowerShell Runspaces

You saw runspaces in [chapter 12](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-12-102.xhtml#ch12) when we discussed PowerShell workflows. We’ll cover runspaces from an API viewpoint in [chapter 20](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-20-150.xhtml#ch20), but for now think of a runspace as an instance of PowerShell running inside an application. The PowerShell console and the ISE are runspaces you’re already familiar with.

Runspaces are useful in cases where you need high-performance parallel processing, so being able to debug the scripts inside a runspace will be useful. This is available only in PowerShell v5.

In a newly opened PowerShell console, try the following:

PS> Get-Runspace

Id Name ComputerName Type State Availability

-- ---- ------------ ---- ----- ------------

1 Runspace1 localhost Local Opened Busy

You’ll see the identical result in a newly opened instance of the ISE.

Create and open a new runspace:

PS> $rsp = [RunspaceFactory]::CreateRunspace()

PS> $rsp.Open()

#### Note

RunspaceFactory is a shortcut to System.Management.Automation.Runspaces.RunspaceFactory.

Try Get-RunSpace again:

PS> Get-Runspace

Id Name ComputerName Type State Availability

-- ---- ------------ ---- ----- ------------

1 Runspace1 localhost Local Opened Busy

2 Runspace2 localhost Local Opened Available

Now you have two runspaces. The original runspace shows as busy—it’s the console (or the ISE) that you’re using. The new runspace shows as available, so let’s get it working:

PS> $ps = [powershell]::Create()

PS> $ps.Runspace = $rsp

PS> [void]$ps.AddScript('C:\test\dbjob.ps1')

PS> $as = $ps.BeginInvoke()

Create an instance of PowerShell and set its runspace to the new runspace. Add a script to the runspace (it’s the same one as in previous examples in this section) and use the BeginInvoke() method to run the script asynchronously.

#### Note

PowerShell is a shortcut for System.Management.Automation.PowerShell.

Now that the script is running, you can see that both runspaces are busy:

PS> Get-Runspace

Id Name ComputerName Type State Availability

-- ---- ------------ ---- ----- ------------

1 Runspace1 localhost Local Opened Busy

2 Runspace2 localhost Local Opened Busy

You can now attach the debugger to the runspace:

PS> Debug-Runspace -Id 2

Debugging Runspace: Runspace2

To end the debugging session type the 'Detach' command at

the debugger prompt, or type 'Ctrl+C' otherwise.

Entering debug mode. Use h or ? for help.

At C:\test\dbjob.ps1:2 char:8

+ while ($true) {

+ ~~~~~

[DBG]: [Process:4244]: [Runspace2]: PS C:\WINDOWS\system32>>

You can now perform debugging tasks such as viewing variable contents:

[DBG]: [Process:4244]: [Runspace2]: PS C:\WINDOWS\system32>> $i

69

[DBG]: [Process:4244]: [Runspace2]: PS C:\WINDOWS\system32>>

Notice that the debug prompt shows you the process and the runspace you’re debugging. Use detach to exit the debugger and allow the script to continue. In this case your script is an infinite loop, so it needs to be stopped:

PS> $ps.Stop()

PS> Get-Runspace

Id Name ComputerName Type State Availability

-- ---- ------------ ---- ----- ------------

1 Runspace1 localhost Local Opened Busy

2 Runspace2 localhost Local Opened Available

Close and remove the runspace:

PS> $rsp.Close()

PS> $rsp.Dispose()

If your runspace is hosted in a different process (application), you can use Enter -PSHostProcess to start debugging. On remote systems, enable Enter-PSHostProcess on the remote machine and connect to the process from within a remote PowerShell session. First, identify the PowerShell host to which you want to connect:

PS> Get-PSHostProcessInfo

ProcessName ProcessId AppDomainName MainWindowTitle

----------- --------- ------------- ---------------

powershell 4244 DefaultAppDomain C:\test

powershell\_ise 4624 DefaultAppDomain C:\test

You can then connect to the relevant host. In this case we’re connecting to the ISE from the PowerShell console:

PS> Enter-PSHostProcess -Id 4624

[Process:4624]: PS C:\Users\Richard\Documents>

The prompt changes to show you’ve connected to a particular process. If the PowerShell host to which you need to connect is on a remote system, enter a PowerShell remoting session to that system and then use Enter-PSHostProcess. In either case, use Debug-Runspace, as shown earlier, to perform your debugging. Exit-PSHostProcess is used to leave the PowerShell runspace.

## 15.6 Summary

* The Write-\* cmdlets can provide diagnostic information during script execution.
* The preference variables control the output of the Write-\* cmdlets.
* Scripts can write diagnostic information to the event logs.
* You can create your own event log to store information from PowerShell scripts.
* Strict mode captures some errors before the script runs.
* Use the latest version of strict mode for maximum effect.
* Scripts can be statically analyzed for errors.
* PSScriptAnalyzer contains many rules that may identify problems in your code.
* Transcripts of the output of a PowerShell session are now available from the ISE and other hosts as well as the console.
* The PowerShell debugger is available in the console and the ISE.
* The PowerShell debugger uses the same keyboard shortcuts as Visual Studio.
* You can execute non-debugger commands in the debugger because the debugger is a full reentrant PowerShell session.
* You can set a breakpoint in a script to force entry to the debugger when it’s reached based on arbitrary conditional logic.
* You can set breakpoints on lines, variables, or commands.
* PowerShell v5 introduces the capability of debugging workflows, PowerShell jobs, scripts on remote machines, and scripts in other runspaces.

You’ve seen how to manage errors and debug your code in the last two chapters. In the next chapter, you’ll start to put PowerShell to use when we investigate how to work with PowerShell data providers, files, and CIM classes.

## Chapter 16: Working with Providers, Files, and CIM

### Overview

This chapter covers

* PowerShell providers
* Files, text, and XML
* Accessing COM objects
* Using CIM

*Outside of a dog, a book is man’s best friend. Inside of a dog, it’s too dark to read.  
Groucho Marx*

No matter how hard you try to avoid it, you’ll have to work with data at some time while using PowerShell. The great news is that PowerShell can work with data in about any format you care to name. The not-so-good news is that you’ll have to learn a bunch of new techniques to work with that data.

In this chapter, we’re going to concentrate on using PowerShell to

* Work with flat files, including XML
* Access COM objects
* Use the Common Information Model (CIM) classes to perform administration tasks

In addition, PowerShell can expose data stores, such as the registry, SQL Server, or Active Directory, in the same way as it exposes the file system through PowerShell providers. This means that once you’ve learned how to work with one provider you have a minimal learning curve to work with the others. This is huge boost to your productivity and makes you immediately effective with new technologies.

We’ll start by looking at the providers built in to PowerShell before moving on to the other types of data.

## 16.1 PowerShell Providers

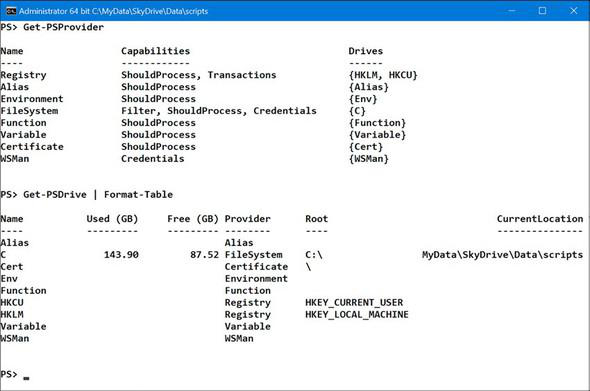
PowerShell does a lot of work to promote a consistent user experience when navigating through hierarchical namespaces. This consistency allows you to use the same commands to work with the file system, the registry, and other stores. The core mechanism that PowerShell uses to accomplish this is the PowerShell provider model. A PowerShell provider is a software component, loaded through modules or snap-ins, that’s used to produce a file system–like experience for other data stores, such as the registry.

#### Note

Providers can’t be written in PowerShell; you have to use C# or another compiled language.

PowerShell installs a number of providers by default. You can view the installed providers by using Get-PSProvider, as shown in [figure 16.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-161-126.xhtml#ch16fig01).

Figure 16.1: The default PowerShell providers and PSDrives



A provider will expose a data store as one or more named drives. These are referred to as PSDrives to avoid confusion with the system drives. You can use Get-PSDrive to view the available drives, also shown in [figure 16.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-161-126.xhtml#ch16fig01).

We’ll look at PSDrives in more detail later in this section, but first we need to deal with the cmdlets that you use to work the data exposed through PSDrives.

### 16.1.1 PowerShell Core Cmdlets

A PowerShell provider is an installable component usually packaged as part of a PowerShell module or snap-in. The basic architecture of the provider module aligns with what are called the core cmdlets. These cmdlets provide the common (or core) activities and are grouped by noun: Item, ChildItem, ItemProperty, Content, Location, Path, PSDrive, and PSProvider, as shown in [table 16.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-161-126.xhtml#ch16table01).

#### Note

The help files about\_Core\_Commands and about\_providers supply further information.

| **Table 16.1: PowerShell core cmdlets** | | |
| --- | --- | --- |
| **Noun** | **Cmdlets** | **Purpose** |
| Item | Clear-Item; Copy-Item; Get-Item; Invoke-Item; Move-Item; New-Item; Remove-Item; Rename-Item; Set-Item | Work directly with an item in a data store |
| ChildItem | Get-ChildItem | Access and manipulate items that are children of a particular item |
| ItemProperty | Clear-ItemProperty; Copy-ItemProperty; Get-ItemProperty; Move-ItemProperty; New-ItemProperty; Remove-ItemProperty; Rename-ItemProperty; Set-ItemProperty | Access and manipulate the properties of an item |
| Content | Add-Content; Clear-Content; Get-Content; Set-Content | Access and manipulate content of text files |
| Location | Get-Location; Pop-Location; Push-Location; Set-Location | Access and manipulate location within a PSDrive |
| Path | Convert-Path; Join-Path; Resolve-Path; Split-Path; Test-Path | Access and manipulate, paths within a PSDrive |
| PSDrive | Get-PSDrive; New-PSDrive; Remove-PSDrive | Access and manage PSDrives |
| PSProvider | Get-PSProvider | Access PowerShell providers |

Many of the core cmdlets have dynamic parameters, that is, parameters that are only available in one or more PSDrives. A help file is available for each of the built-in providers that details the dynamic parameters on each core cmdlet. These can be accessed using the Provider name. For example:

PS> Get-Help Registry

Providers are the heart of the namespace mechanism, but you don’t usually work directly with them. Instead, you work through named drives that allow you access to the provider’s capabilities.

### 16.1.2 Working with PSDrives

PowerShell providers are typically accessed through named drives. This means that each provider will have at least one drive associated with it. The drives that a provider exports needn’t correspond to things like system disk drives (though the file system provider usually has one drive name exported for each physical drive on the computer). Their names can also be longer than the single character permitted in drive letters.

The provider-exported named drives are called PSDrives. Similarly, a path that contains a PSDrive is called a PSPath, and a path that contains a physical drive is called a provider-specific path. A PSPath must be translated into the provider-specific path form before it can be processed by the system.

Another useful feature supported by many providers is the ability to create your own drive names. That means you can, for example, create a PSDrive as a shortcut to a common resource such as a test folder on your machine:

PS> New-PSDrive -Name Test -PSProvider FileSystem `

-Description 'Test area' -Root C:\test\

Name Used (GB) Free (GB) Provider Root CurrentLocation

---- --------- --------- -------- ---- ---------------

Test 0.00 89.62 FileSystem C:\test\

You’re able to access this drive only from the PowerShell session in which you created it. If you need this drive in all sessions, add the creation command to your profile. You can create persistent mapped network drives using the -Persist parameter.

#### Accessing the documents folder

If you want to work with folders in your user area, use the ~ as a shortcut. For instance:

Get-ChildItem -Path (Resolve-Path -Path ~)

Get-ChildItem -Path (Resolve-Path -Path ~\documents)

That will access your user area and your documents folder, respectively. The call to Resolve-Path converts the PSPath to an absolute provider path. Note that ~ refers to the home directory of the current provider, so if you’ve performed a change directory action into the Registry, ~ won’t refer to your user area any longer.

You can move into the new drive or access it using the core cmdlets from [table 16.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-161-126.xhtml#ch16table01). PowerShell enables you to create appropriate items in the new drive (you couldn’t create a file if you’d used the registry provider for the new drive) and access the content of those items:

PS> Add-Content -Value 'Hello There' -Path test:\junk.txt

PS> Get-Content -Path Test:\junk.txt

Hello There

#### Using non-PowerShell applications with a PSDrive

Non-PowerShell applications don’t understand PSDrives and therefore can’t use paths containing PSDrives directly. If you’re in the PSDrive, the system auto-matically sets the current directory properly on the child process object to the provider-specific path before starting the process. You need to use the provider-specific path as supplied by Resolve-Path:

cmd /c type (Resolve-Path test:/junk.txt).ProviderPath

rather than

cmd /c type test:/junk.txt

PowerShell has another couple of tricks for working with paths.

### 16.1.3 Working with Paths

Most of the time paths work, but there are special cases to consider.

* Hidden files aren’t normally displayed by the file system provider; force is required to see hidden files.
* The PowerShell provider infrastructure has universal support for wildcards, though [ and ] need special care.
* The -LiteralPath parameter suppresses pattern-matching, which makes dealing with paths containing wildcard characters much easier.

#### Hidden Files

By default, Get-ChildItem doesn’t show hidden files, and the item files won’t access them either. You need to use the -Force parameter. Try this:

PS> Get-ChildItem -Path C:\ -Filter \*.sys

You’ll see nothing returned. Use the –Force parameter

PS> Get-ChildItem -Path C:\ -Filter \*.sys -Force

Directory: C:\

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a-hs- 15/05/2017 09:37 6843224064 hiberfil.sys

-a-hs- 12/05/2017 14:10 2550136832 pagefile.sys

-a-hs- 12/05/2017 14:10 16777216 swapfile.sys

and everything is visible.

The Get-ChildItem cmdlet also has the -Attributes, -Hidden, -ReadOnly, and -System dynamic parameters on the file system provider for dealing with particular file attributes. The file system provider also supports the -Directory and -File dynamic parameters for restricting output to directories and files, respectively.

#### Paths and Wildcards

You can use wildcards any place you can navigate to, even in places such as the PSDrive you created earlier:

PS> Get-ChildItem -Path test:\\*.txt | Format-Table -AutoSize

Directory: C:\test

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 31/01/2017 20:09 57 ac.txt

-a---- 16/05/2016 10:46 52 data.txt

-a---- 09/05/2017 19:54 804 err.txt

-a---- 08/02/2016 18:52 34 My New File [1].txt

-a---- 08/02/2016 18:52 34 My New File [2].txt

-a---- 08/02/2016 18:52 34 My New File [3].txt

-a---- 15/05/2017 11:11 13 junk.txt

We might all agree that this is a great feature, but there’s a down side. Suppose you want to access a path that contains one of the wildcard metacharacters: ?, \*, [, and ]. In the Windows file system, \* and ? aren’t a problem because you can’t use these characters in a filename or directory name. But you can use [ and ]. Working with files whose names contain [ or ] can be quite a challenge because of the way wildcards and quoting work. Square brackets are used a lot in filenames that applications create where they avoid collisions by numbering the files. Some examples are shown in the output immediately preceding this.

If you want only the files that contain [ or ], you need some special processing because the [ is being treated as part of a wildcard pattern. Clearly you need to suppress treating [ as a wildcard by quoting it. The backtick is the obvious candidate, but a single backtick is insufficient. If you keep adding backticks, you’ll eventually get a result:

PS> Get-ChildItem -Path test:\\*````[\*.txt

Directory: C:\test

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 08/02/2016 18:52 34 My New File [1].txt

-a---- 08/02/2016 18:52 34 My New File [2].txt

-a---- 08/02/2016 18:52 34 My New File [3].txt

But if you want all the numbered versions of a particular file, you’ll end up with something like this:

PS> Get-ChildItem -Path "test:\My New File ````[\*````].txt"

You have to use double quotes and four backticks each for the [ and ]. Much of the complication arises because you want some of the metacharacters to be treated as literal characters, whereas the rest still do pattern-matching. Trial and error is usually the only way to get this right.

#### Note

As we’ve said before, this stuff is hard. It’s hard to understand and it’s hard to get right. Unfortunately, no one has yet to come up with a better mechanism. This problem occurs in any language that supports pattern-matching. Patience, practice, and experimentation are the only ways to figure it out.

You can avoid a lot of the trial and error by using the -LiteralPath parameter.

#### The -LiteralPath Parameter

The -LiteralPath parameter is available on most core cmdlets. Say you want to copy a file from the previous example. If you use the regular path mechanism in Copy-Item

PS> Set-Location -Path C:\test\

PS> Copy-Item 'My New File [1].txt' C:\test1\junk.txt

PS> Get-ChildItem -Path C:\test1\junk.txt

Get-ChildItem : Cannot find path 'C:\test1\junk.txt' because it does not exist.

At line:1 char:1

+ Get-ChildItem -Path C:\test1\junk.txt

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound: (C:\test1\junk.txt:String) [Get-ChildItem], ItemNotFoundException

+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand

the copy fails because the square brackets were treated as pattern-matching metacharacters. Now try it using -LiteralPath:

PS> Copy-Item -LiteralPath 'My New File [1].txt' C:\test1\junk.txt

PS> Get-ChildItem -Path C:\test1\junk.txt

Directory: C:\test1

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 08/02/2016 18:52 34 junk.txt

This time it works properly.

#### PowerShell 5.0 enhancement

In PowerShell v5 on Windows 10, the PSReadline module is automatically loaded and brings a number of enhancements to the PowerShell console. One enhancement deals with escaping metacharacters in filenames. If you type

PS>Remove-Item -Path C:\test1\junk.txt

PS>Copy-Item my

and then press the Tab key, you’ll find that the filename is expanded with the required escape characters:

PS>Copy-Item '.\My New File `[1`].txt'

You can then perform the copy:

PS> Copy-Item '.\My New File `[1`].txt' C:\test1\junk.txt

PS> Get-ChildItem -Path C:\test1\junk.txt

Directory: C:\test1

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 08/02/2016 18:52 34 junk.txt

The -LiteralPath parameter is still necessary for PowerShell versions 4.0 and earlier.

When you pipe the output of a cmdlet such as Get-ChildItem into another cmdlet like Remove-Item, the -LiteralPath parameter is used to couple the cmdlets so that metacharacters in the paths returned by Get-ChildItem (including aliases dir or ls) don’t cause problems for Remove-Item. If you want to delete the files we were looking at earlier, you can use Get-ChildItem to see them:

PS> Get-ChildItem 'My New File ``[\*'

Directory: C:\test

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 23/01/2016 17:37 12 My New File [1].txt

-a---- 23/01/2016 17:37 12 My New File [2].txt

-a---- 23/01/2016 17:37 12 My New File [3].txt

Now pipe the output of Get-ChildItem into Remove-Item

PS> Get-ChildItem 'My New File ``[\*' | Remove-Item

and verify that they’ve been deleted.

This covers the issues around working with file paths. From here we can move on to working with the file contents after a quick look at the Registry provider.

### 16.1.4 The Registry Provider

PowerShell uses paths to access many types of hierarchical information on a Windows computer. Probably the most important type of hierarchical information is the Registry, a store of hierarchical configuration information, much like the file system. But there’s one significant difference—in the Registry, a container has two axes: children and properties or, as you’re more used to calling them from hashtables, keys and values. This is one of the more complex scenarios that the provider model addresses.

In the Registry provider, it’s no longer sufficient to have only the path; you also need to know whether you’re accessing a name or a property. Let’s take a look. Start by cd’ing to the PowerShell hive in the Registry:

PS> cd hklm:\software\microsoft\powershell

Let’s see what’s there:

PS> Get-ChildItem

Hive: HKEY\_LOCAL\_MACHINE\software\microsoft\powershell

Name Property

---- --------

1 Install : 1...

3 ConsoleHostShortcutTarget : C:\..

Unfortunately, the default display for a Registry entry is a bit cryptic, and for once using Format-List doesn’t make it any more comprehensible. But you found an item named 1, which we can dig into:

PS> Get-ChildItem ./1

Hive: HKEY\_LOCAL\_MACHINE\software\microsoft\PowerShell\1

Name Property

---- --------

0409 Install : 1

PowerShellEngine ApplicationBase : C:\Win...

PSConfigurationProviders

ShellIds

You see information about the subkeys, but what about accessing the properties? First, you need to determine if there are any subkeys:

PS> Get-Item ./1 | select Property

Property

--------

{Install, PID}

You then use the Get-ItemProperty cmdlet to access the property:

PS> Get-ItemProperty -Path ./1 -Name PID

PID : 89383-100-0001260-04309

PSPath : Microsoft.PowerShell.Core\Registry::

HKEY\_LOCAL\_MACHINE\software\microsoft\

PowerShell\1

PSParentPath : Microsoft.PowerShell.Core\Registry::

HKEY\_LOCAL\_MACHINE\software\microsoft\

PowerShell

PSChildName : 1

PSDrive : HKLM

PSProvider : Microsoft.PowerShell.Core\Registry

Notice that you need to specify both the path and the name of the property to retrieve. Properties are always relative to a path. There’s another somewhat annoying thing about how Get-ItemProperty works: It doesn’t return the value of the property—it returns a new object that has the property value as a member. Before you can do anything with this value, you need to extract it from the containing object:

PS> (Get-ItemProperty -Path ./1 -Name PID).PID

89383-100-0001260-04309

By using the . operator to extract the member’s value, you can get the value. You could also use this:

PS> Get-ItemProperty -Path ./1 -Name PID | select -ExpandProperty PID

89383-100-0001260-04309

#### Note

This is another one of those design trade-offs the PowerShell team encountered as they developed this environment. If only the value was returned, you’d lose the context for the value (where it came from and so on). In order to preserve this information, the team ended up forcing people to write what appears to be redundant code. A better way to handle this might’ve been to return the value with the context attached as synthetic properties.

So far, we’ve looked at accessing the HKEY\_CURRENT\_USER and HKEY\_LOCAL\_MACHINE Registry hives. These are the only two hives for which PowerShell drives are created by default. How do you access the other hives? The answer is to fall back on the provider:

PS> Get-ChildItem -Path Registry::

Hive:

Name Property

---- --------

HKEY\_LOCAL\_MACHINE

HKEY\_CURRENT\_USER

HKEY\_CLASSES\_ROOT EditFlags : {0, 0, 0, 0}

HKEY\_CURRENT\_CONFIG

HKEY\_USERS

HKEY\_PERFORMANCE\_DATA Global : {80, 0, 69, 0...}

Costly : {80, 0, 69, 0...}

You can follow the paths through the Registry starting with the provider rather than a drive:

PS> Get-ChildItem -Path registry::HKEY\_CURRENT\_CONFIG\System\CurrentControlSet\SERVICES\TSDDD\

Hive: HKEY\_CURRENT\_CONFIG\System\CurrentControlSet\SERVICES\TSDDD

Name Property

---- --------

DEVICE0 Attach.ToDesktop : 1

#### Note

Other PowerShell drives are also accessible via their providers using similar syntax.

Now that you’re more familiar with PowerShell providers, let’s look at how you can work with files and their content.

## 16.2 Files, Text, and XML

You saw earlier in the chapter how to work with the file system provider. In this section, we’ll show you how to read and write file content. We’ll then cover how to work with unstructured text as well as XML structured text. Let’s start by discovering how to read file content and how to write to files.

### 16.2.1 File Processing

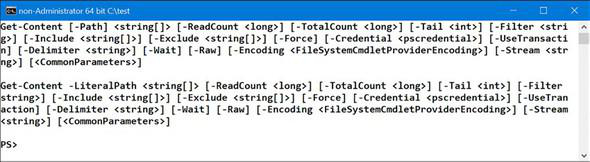
In PowerShell, files are read using the Get-Content cmdlet. This cmdlet allows you to work with text files using a variety of character encodings and lets you work efficiently with binary files. Writing files is a bit more complex, because you have to choose between Set-Content (or Add-Content) and Out-File. The difference here is whether the output goes through the formatting subsystem.

It’s important to point out that there are no separate open/read/close or open/write/close steps to working with files. The pipeline model allows you to process data and never have to worry about closing file handles—the system takes care of this for you.

#### Reading Text Files

The Get-Content cmdlet is the primary way to read files in PowerShell. In fact, it’s the primary way to read any content available through PowerShell drives. [Figure 16.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16fig02) shows this cmdlet’s syntax.

Figure 16.2: Get-Content syntax



Reading text files is simple. The command

PS> Get-Content -Path myfile.txt

will send the contents of myfile.txt to the output stream. Notice that the command signature for -Path allows for an array of path names. This is how you concatenate a collection of files. Let’s try this. First, create a collection of files:

PS> Set-Location -Path C:\test\

PS> 1..3 | foreach { "This is file $\_" > "file$\_.txt"}

And now display their contents:

PS> Get-Content -Path file1.txt,file2.txt,file3.txt

This is file 1

This is file 2

This is file 3

#### Note

PowerShell uses Unicode encoding by default. If you want to read PowerShell-created files using cmd.exe utilities, you have to use ASCII encoding.

#### Heads or tails?

You’ve seen that the -TotalCount parameter can be used to control how many lines are read from a file. This parameter has an alias of -Head to fit with other file-reading utilities (Get-Content has aliases of type and cat). Create a file with a number of lines:

PS> 1..10 | foreach { "This is line $\_" |

Add-Content -Path multifile.txt}

You can read the beginning of the file:

PS> Get-Content -Path .\multifile.txt -Head 3

or you can read the end of the file:

PS> Get-Content -Path .\multifile.txt -Tail 3

This won’t work though:

PS> Get-Content -Path .\multifile.txt -Head 3 -Tail 3

-Head and -Tail are mutually exclusive.

Get-Content, by default, reads a line at a time, so you get an array of lines if you do this:

PS> $v1 = Get-Content .\multifile.txt

PS> $v1.Count

10

If you want the text in the file to be a single string, you use the -Raw parameter:

PS> $v2 = Get-Content .\multifile.txt -Raw

PS> $v2.count

1

That’s about it for text files. Reading binary files takes a little more work than simple text files.

#### Reading Binary Files

The function in the next listing can be used to display the contents of a binary file. It takes the name of the file to display, the number of bytes to display per line, and the total number of bytes as parameters.

#### Listing 16.1: Get-HexDump

function Get-HexDump {

param (

[Parameter(Mandatory)]

[string]$path,

[int]$width=10,

[int]$total=-1

)

$OFS='' 1

Get-Content -Encoding byte -Path $path -ReadCount $width `

-TotalCount $total |

foreach {

$record = $\_

if (($record -eq 0).count -ne $width) 2

{

$hex = $record | foreach { 3

' ' + ('{0:x}' -f $\_).PadLeft(2,'0')} 3

$char = $record | foreach{ 3

if ([char]::IsLetterOrDigit($\_)) 3

{ [char] $\_ } else { '.' }} 3

"$hex $char"

}

}

}

* 1 Set $OFS to empty string
* 2 Skip record if length is zero
* 3 Format data

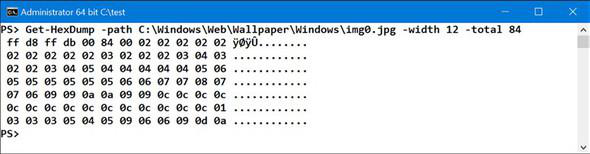
The function takes a mandatory path parameter and optional parameters for the number of bytes per line and the total number of bytes to display. You’re going to be converting arrays to strings and you don’t want any spaces added, so you’ll set the output field separator character 1 to be empty.

The Get-Content cmdlet does all the hard work. It reads the file in binary mode (indicated by setting -Encoding to byte), reads up to a maximum of -TotalCount bytes, and writes them into the pipeline in records of length specified by -ReadCount. The first thing you do in the foreach scriptblock is save the record that was passed in, because you’ll be using nested scriptblocks that will cause $\_ to be overwritten.

If the record is all zeros 2, you won’t bother displaying it. It might be a better design to make this optional, but we’ll leave it as is for this example. For display purposes, you’re converting the record of bytes 3 into two-digit hexadecimal numbers. You use the format operator to format the string in hexadecimal and then the PadLeft() method on strings to pad it out to two characters. Finally, you prefix the whole thing with a space. The variable $hex ends up with a collection of these formatted strings.

Now you need to build the character equivalent of the record. You’ll use the methods on the [char] class to decide whether you should display the character or a dot (.). Notice that even when you’re displaying the character, you’re still casting it into a [char]. This is necessary because the record contains a byte value, which, if directly converted into a string, will be formatted as a number instead of as a character. Finally, you’ll output the completed record, taking advantage of string expansion to build the output string (which is why you set $OFS to ‘’). Example output is shown in [figure 16.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16fig03).

Figure 16.3: Example Get-HexDump output



In this example, you’re using Get-HexDump to dump out the contents of one of the Windows bitmap files, specifying that it display 12 bytes per line and stop after the first 84 bytes. The first part of the display is the value of the byte in hexadecimal, and the portion on the right side is the character equivalent. Only values that correspond to letters or numbers are displayed. Nonprintable characters are shown as dots.

#### Note

You may find that PowerShell performs slowly when reading large files. One option in that case is to use the .NET I/O classes.

That covers reading files, but what about writing to a file?

#### Writing Files

You have two major ways to write files in PowerShell—by setting file content with the Set-Content cmdlet and by writing files using the Out-File cmdlet. The big difference is that Out-File, like all the output cmdlets, tries to format the output. Set-Content writes the output as is. If its input objects aren’t already strings, it will convert them to strings by calling the ToString() method. This isn’t usually what you want for objects, but it’s exactly what you want if your data is already formatted or if you’re working with binary data.

The other thing you need to be concerned with is how the files are encoded when they’re written. In an earlier example, you saw that by default text files are written in Unicode. Let’s rerun this example, changing the encoding to ASCII instead:

PS> 1..3 | foreach{ "This is file $\_" |

Set-Content -Encoding ascii file$\_.txt }

The -Encoding parameter is used to set how the files will be written

#### Note

You use Set-Content rather than Out-File because Out-File adds extra processing overhead that you don’t need when writing primitive data such as text. And be aware that Out-File and Set-Content use different default encodings. A standard default encoding may be used in PowerShell v6.

### 16.2.2 Unstructured Text

Although PowerShell is an object-based shell, it still has to deal with text. In [chapter 3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-3-24.xhtml#ch03), we covered the operators that PowerShell provides for working with text. In this section, we’ll cover some of the more advanced string-processing operations. We’ll discuss techniques for splitting and joining strings using the [string] and [regex] members and using filters to extract statistical information from a body of text.

#### Using System.String to Work with Text

PowerShell has the -split operator to separate elements of a string. The Split() method on the [string] class provides a few more options. The Split() method with no arguments splits on spaces. In this example, it produces an array of three elements:

PS> 'Hello there world'.Split().length

3

In fact, it splits on any of the characters that fall into the WhiteSpace character class. This includes tabs, so it works properly on a string containing both tabs and spaces:

PS> "Hello`there world".Split()

Hello

here

world

Notice the second element! The characters `t are interpreted as a tab character.

Although the default is to split on a whitespace character, you can specify a string of characters to use to split fields:

PS> 'First,Second;Third'.Split(',;')

First

Second

Third

Here you specify the comma and the semicolon as valid characters to split the field.

There is, however, an issue; the default behavior for “split this” isn’t necessarily what you want. The reason is that it splits on each separator character. This means that if you have multiple spaces between words in a string, you’ll get multiple empty elements in the result array, for example:

PS> 'Hello there world'.Split().length

6

In this example, you end up with six elements in the array because there are three spaces between there and world. Looking at the MSDN documentation for the Split() method, you’ll see that there are options to use StringSplitOptions. You can test what the options do by casting a string into the options:

PS> [StringSplitOptions]'abc'

Cannot convert value "abc" to type "System.StringSplitOptions".

Error: "Unable to match the identifier name abc to a valid

enumerator name. Specify one of the following enumerator names

and try again:

None, RemoveEmptyEntries"

At line:1 char:1

+ [StringSplitOptions]'abc'

+ ~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidArgument: (:) [], RuntimeException

+ FullyQualifiedErrorId : SubstringDisambiguationEnumParseThrewAnException

RemoveEmptyEntries looks like it might solve your problem:

PS> 'Hello there world'.split(' ',[StringSplitOptions]::RemoveEmptyEntries)

Hello

there

world

It works as desired. Next, you can apply this technique to a larger problem.

#### Analyzing Word Use in a Document

Given a body of text, say you want to find the number of words in the text as well as the number of unique words and then display the 10 most common words in the text. For our purposes, we’ll use one of the PowerShell help text files: about\_Assignment\_operators.help.txt. Remember that Get-Content creates an array where each element is a line from the file. You want a single string so you have to use the -Raw parameter:

PS> $s = Get-Content `

-Path $PSHOME\en-US\about\_Assignment\_Operators.help.txt `

-Raw

PS> $s.length

22780

$s contains a single string containing the whole text of the file. Next, split it into an array of words:

PS> $words = $s.Split(" `t", [stringsplitoptions]::RemoveEmptyEntries)

PS> $words.Length

3453

The text of the file has 3453 words in it. You need to find out how many unique words there are. The easiest approach is to use the Sort-Object cmdlet with the -Unique parameter. This code will sort the list of words and then remove all the duplicates:

PS> $uniq = $words | sort -Unique

PS> $uniq.count

719

The help topic contains 719 unique words. Using the Sort-Object cmdlet is fast and simple, but it doesn’t give the frequency of use. Luckily, PowerShell includes a cmdlet that’s useful for this kind of task: Group-Object. This cmdlet groups its input objects into collections sorted by the specified property. This means you can achieve the same type of ordering with the following:

PS> $grouped = $words | group | sort count

The most frequently used word is, unsurprisingly, “the”:

PS> $grouped[-1]

Count Name Group

----- ---- -----

335 the {the, the, the, the...}

You can display the 10 most frequent words with this:

PS> $grouped[-1..-10]

Count Name Group

----- ---- -----

335 the {the, the, the, the...}

134 to {to, to, to, to...}

121 a {a, a, a, a...}

110 ... {...

97 value {value, value, value, value...}

94 $a {$a, $a, $a, $a...}

85 C:\PS> {C:\PS>, C:\PS>, C:\PS>, C:\PS>...}

80 of {of, of, of, of...}

74 = {=, =, =, =...}

55 variable {variable, variable, variable...}

The code creates a nicely formatted display courtesy of the formatting and output subsystem built into PowerShell. In the world of unstructured text, you’ll quickly run into examples where simple splits aren’t enough. As is so often the case, regular expressions come to the rescue.

Regular expressions are a domain-specific language (DSL) for matching and manipulating text. We covered a number of examples using regular expressions with the -match and -replace operators in [chapter 3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-3-24.xhtml#ch03). This time, you’re going to work with the regular expression class itself.

#### Splitting Strings with Regular Expressions

There’s a type accelerator, [regex], for the regular expression type. The [regex] type also has a Split() method, but it’s much more powerful because it uses a regular expression to decide where to split strings instead of a single character:

PS> $s = "Hello-1-there-22-World!"

PS> [regex]::split($s,'-[0-9]+-')

Hello

there

World!

In this example, the fields are separated by a sequence of digits bound on either side by a dash. This pattern couldn’t be specified with simple character-based split operations.

When working with the .NET regular expression library, the [regex] class isn’t the only class that you’ll run into. You’ll see this in the next example, when we look at using regular expressions to tokenize a string.

#### Tokenizing Text with Regular Expressions

Tokenization, or the process of breaking a body of text into a stream of individual symbols, is a common activity in text processing; for instance, the PowerShell interpreter has to tokenize a script before it can be executed. In the next example, we’re going to look at how you can write a simple tokenizer for basic arithmetic expressions you might find in a programming language. First, you need to define the valid tokens in these expressions. You want to allow numbers made up of one or more digits; allow expressions made up of any of the operators +, -, \*, or /; and also allow sequences of spaces. Here’s what the regular expression to match these elements looks like:

PS> $pat = [regex] "[0-9]+|\+|\-|\\*|/| +"

This is a pretty simple pattern using only the alternation operator | and the quantifier +, which matches one or more instances. Because you used the [regex] cast in the assignment, $pat contains a regular expression object. You can use this object directly against an input string by calling its Match() method:

PS> $m = $pat.match("11+2 \* 35 -4")

The Match() method returns a Match object (the full type name is System.Text.RegularExpressions.Match). You can use the Get-Member cmdlet to explore the full set of members on this object at your leisure, but for now you’re interested in only three members. The first member is the Success property. This will be true if the pattern matched. The second interesting member is the Value property, which will contain the matched value. The final member you’re interested in is the NextMatch() method. Calling this method will step the regular expression engine to the next match in the string and is the key to tokenizing an entire expression. You can use this method in a while loop to extract the tokens from the source string one at a time. In the example, you keep looping as long as the Match object’s Success property is true. Then you display the Value property and call NextMatch() to step to the next token:

PS> while ($m.Success)

{

$m.value

$m = $m.NextMatch()

}

11

+

2

\*

35

-

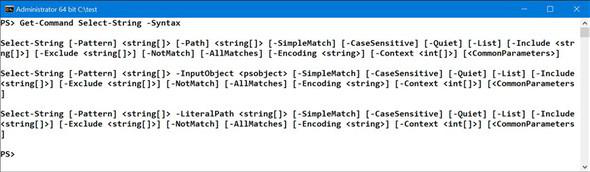
4

In the output, you see each token, one per line, in the order in which they appeared in the original string.

#### Searching Files with the Select-String Cmdlet

The Select-String cmdlet allows you to search through collections of strings or collections of files. It’s similar to the grep command on UNIX-derived systems and the findstr command on Windows. [Figure 16.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16fig04) shows the parameters on this cmdlet. Select-String is optimized for searching through files.

Figure 16.4: Syntax of Select-String



Let’s search through all of the “about\_\*” topics in the PowerShell installation directory to see if the phrase “wildcard description” is there:

PS> Select-String -Path $pshome/en-US/about\*.txt `

-Pattern 'wildcard description'

C:\Windows\System32\WindowsPowerShell\v1.0\en-US\about\_Wildcards.help.txt:21:

Wildcard Description Example

Match No match

You see that there’s exactly one match, but notice the uppercase letters in the matching string. If you rerun the search using the -CaseSensitive parameter

PS> Select-String -Path $pshome/en-US/about\*.txt `

-Pattern 'wildcard description' -CaseSensitive

nothing is found. Searching through files this way can sometimes produce more results than you need.

Normally Select-String will find all matches in a file. The -List switch limits the search to the first match in a file. The -Quiet switch returns $true if any of the files contain a match and $false if none do. You can also combine the two switches so that the cmdlet returns the first match in the set of files.

If you want to search a more complex set of files, you can pipe the output of Get-ChildItem into the cmdlet and it will search all of these files. Let’s search all the log files in the system32 subdirectory:

PS> Get-ChildItem -Recurse -Filter \*.log -Path $env:windir\system32 |

Select-String -List -Pattern 'fail' | Format-Table path

Only the path to a log file containing a record with “fail” in it will be displayed.

The MatchInfo object produced by Select-Object has a context property. This property allows you to have Select-String include the lines before and after the matching line. You can specify two numbers to the -Context parameter. The first number is the length of the prefix context and the second is the suffix context; for instance, to get only the matching line and the four following lines, you have to specify a prefix context of 0 and a suffix of 4:

PS> Get-Help Select-String |

Out-String -Stream |

Select-String -Pattern 'syntax' -Context 0,4

> SYNTAX

Select-String [-Pattern] <String[]> [-Path] <String[]> [-AllMatches]

[-CaseSensitive] [-Context <Int32[]>] [-Encoding

{unicode | utf7 | utf8 | utf32 | ascii |

bigendianunicode |default | oem}]

[-Exclude <String[]>] [-Include <String[]>]

[-InformationAction {SilentlyContinue | Stop | Continue |

Inquire | Ignore | Suspend}] [-InformationVariable <System.String>]

[-List] [-NotMatch] [-Quiet] [-SimpleMatch] [<CommonParameters>]

#### Getting All Matches in the Line

Another property on the MatchInfo object is the Matches property. This property is used when the -AllMatches switch is specified to the cmdlet. It causes all matches in the line to be returned instead of only the first match. You’ll use this switch to perform the same type of tokenization that you did with regular expressions in [section 16.2.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16lev2sec6). You’ll pipe the expression string into Select-String with the -AllMatches switch and the same regular expression you used earlier:

PS> '1 + 2 \*3' |

Select-String -AllMatches -Pattern '[0-9]+|\+|\-|\\*|/| +' |

foreach { $\_.Matches } | Format-Table -AutoSize

Groups Success Captures Index Length Value

------ ------- -------- ----- ------ -----

{1} True {1} 0 1 1

{ } True { } 1 1

{+} True {+} 2 1 +

{ } True { } 3 1

{2} True {2} 4 1 2

{ } True { } 5 1

{\*} True {\*} 6 1 \*

{3} True {3} 7 1 3

You use the Foreach-Object cmdlet to isolate the Matches property and then format the output as a table. You can see each of the extracted tokens in the Value field in the Matches object. Using this mechanism, you can effectively and efficiently process things like large log files where the output is formatted as a table.

#### Note

If you’re using PowerShell v5.1 on Windows 10 with the Creators update (build version 10.0.15063.413 or later) you’ll see {0} for all entries in the Groups and Captures columns.

All the text so far in this chapter has been unstructured text where there’s no rigorously defined layout for that text. As a consequence, you’ve had to work fairly hard to extract the information you want out of this text. There are, however, large bodies of structured text, where the format is well defined in the form of XML documents.

### 16.2.3 XML Structured Text Processing

XML (Extensible Markup Language) is used for everything from configuration files to log files to databases. PowerShell uses XML for its type and configuration files as well as for the help files. For PowerShell to be effective, it has to be able to process XML documents effectively. Let’s look at how XML is used and supported in PowerShell.

#### Note

This section assumes you possess some basic knowledge of XML markup.

We’ll look at the XML object type, as well as the mechanism that .NET provides for searching XML documents.

#### Using XML as Objects

PowerShell supports XML documents as a core data type. This means that you can access the elements of an XML document as though they were properties on an object. For example, let’s create a simple XML object. Start with a string that defines a top-level node called top. This node contains three descendants: a, b, and c, each of which has a value. Let’s turn this string into an object:

PS> $d = [xml] '<top><a>one</a><b>two</b><c>3</c></top>'

The [xml] cast takes the string and converts it into an XML object of type System.XML.XmlDocument. This object is then adapted by PowerShell so you can treat it as a regular object. Let’s try this out. First, display the object:

PS> $d

top

---

top

As you expect, the object displays one top-level property corresponding to the top-level node in the document. Now let’s see what properties this node contains:

PS> $d.top

a b c

- - -

one two 3

Three properties correspond to the descendants of top. You can use conventional property notation to look at the value of an individual member:

PS> $d.top.a

One

Modifying this node is as simple as assigning a new value to the node. Let’s assign the string “Four” to node a:

PS> $d.top.a = 'Four'

PS> $d.top.a

Four

You can see that it’s been changed. But there’s a limitation: you can only use a string as the node value. The XML object adapter won’t automatically convert non-string objects to strings in an assignment, so you get an error when you try it. All the normal type conversions apply, so you can use a node value in arithmetic actions if it can be converted to a suitable type.

Adding elements to an XML document isn’t a simple assignment operation.

#### Adding Elements to an XML Object

Let’s add an element d to this document. To do so, you need to use the methods on the XML document object. First, you have to create the new element. Then you set the element text, the “inner text,” to a value and finally append the new element to the document:

PS> $el= $d.CreateElement('d')

PS> $el.Set\_InnerText('Hello')

PS> $d.top.AppendChild($el)

Notice that you’re using the property setter method here. This is because the XML adapter hides the basic properties on the XmlNode object. The other way to set this would be to use the PSBase member:

PS> $ne = $d.CreateElement('e')

PS> $ne.InnerText = 'World'

PS> $d.top.AppendChild($ne)

Now that you know how to add children to a node, how can you add attributes? The pattern is the same as with elements. First, create an attribute object. Next, set the value of the text for that object. Finally, add it to the top-level document:

PS> $attr = $d.CreateAttribute('BuiltBy')

PS> $attr.Value = 'Windows PowerShell'

PS> $d.DocumentElement.SetAttributeNode($attr)

You can’t cast the document back to a string and see what it looks like instead; you have to save the document as a file and display it:

PS> $d.Save('C:\test\new.xml')

PS> Get-Content -Path C:\test\new.xml

<top BuiltBy="Windows PowerShell">

<a>one</a>

<b>two</b>

<c>3</c>

<d>Hello</d>

<e>World</e>

</top>

You’ve constructed, edited, and saved XML documents, but you haven’t loaded an existing document yet, so that’s the next step.

#### Loading and Saving XML Files

In the previous section, you saved an XML document to a file and read it, as text, using Get-Content. If you want to work with an XML document, you need to cast the output of Get-Content into an XML document:

PS> $nd = [xml] (Get-Content -Path C:\test\new.xml)

#### Speedier XML reading

By default, Get-Content reads one record at a time. This process can be quite slow. When processing large files, you should use the -ReadCount parameter to specify a block size of –1. Doing so will cause the entire file to be loaded and processed at once, which is much faster. Alternatively, here’s another way to load an XML document using the .NET methods:

PS> ($nd = [xml]'<root></root>').Load('C:\test\new.xml')

Note that this does require that the full path to the file be specified.

Let’s verify that the document was read properly:

PS> $nd.top

BuiltBy : Windows PowerShell

a : one

b : two

c : 3

d : Hello

e : World

Everything is as it should be. Even the attribute is there.

Although this is a simple approach and the one you’ll probably use most often, it’s not necessarily the most efficient approach because it requires loading the entire document into memory. For large documents or collections of documents, loading all the text into memory may become a problem. In the next section, we’ll look at alternative approaches that, though more complex, are more memory efficient.

#### Using the XmlReader Class

Our previously discussed method for loading an XML file is simple but not too efficient. It requires that you load the file into memory, make a copy of the file while turning it into a single string, and create an XML document representing the entire file using the XML Document Object Model (DOM) representation. The DOM allows you to treat an XML document as a hierarchy of objects, but to do so it consumes a lot of memory.

A much more memory-efficient way to process XML documents is to use the System.Xml.XmlReader class. This class streams through the document one element at a time instead of loading the whole thing into memory. You need a function that will use the XML reader to stream through a document and output it properly indented—an XML pretty-printer, if you will.

First, you need a more complex document where there are more attributes and more nesting on which you can test your document.

#### Listing 16.2: Creating the text XML document

@'

<top BuiltBy = "Windows PowerShell">

<a pronounced="eh">

one

</a>

<b pronounced="bee">

two

</b>

<c one="1" two="2" three="3">

<one>

1

</one>

<two>

2

</two>

<three>

3

</three>

</c>

<d>

Hello there world

</d>

</top>

'@ > c:\test\fancy.xml

The function to read XML documents will be called Format-XmlDocument to keep within the PowerShell naming conventions.

#### Listing 16.3: The Format-XmlDocument function

function global:Format-XmlDocument {

param

(

[string]$Path = "$PWD\fancy.xml"

)

$settings = New-Object System.Xml.XmlReaderSettings 1

$doc = (Resolve-Path -Path $Path).ProviderPath

$reader = [System.Xml.XmlReader]::Create($doc, $settings)

$indent=0

function indent {

param

(

[Object]$s

)

' '\*$indent+$s

} 2

while ($reader.Read())

{

if ($reader.NodeType -eq [Xml.XmlNodeType]::Element) 3

{

$close = $(if ($reader.IsEmptyElement) { '/>' } else { '>' })

if ($reader.HasAttributes)

{

$s = indent "<$($reader.Name) "

[void] $reader.MoveToFirstAttribute()

do

{

$s += "$($reader.Name) = `"$($reader.Value)`" "

}

while ($reader.MoveToNextAttribute())

"$s$close"

}

else

{

indent "<$($reader.Name)$close"

}

if ($close -ne '/>') {$indent++} 4

}

elseif ($reader.NodeType -eq [Xml.XmlNodeType]::EndElement )

{

$indent--

indent "</$($reader.Name)>"

}

elseif ($reader.NodeType -eq [Xml.XmlNodeType]::Text)

{

indent $reader.Value 5

}

}

$reader.close()

}

* 1 Create settings object
* 2 Define formatting function
* 3 Process element nodes
* 4 Increase indent level
* 5 Format text element

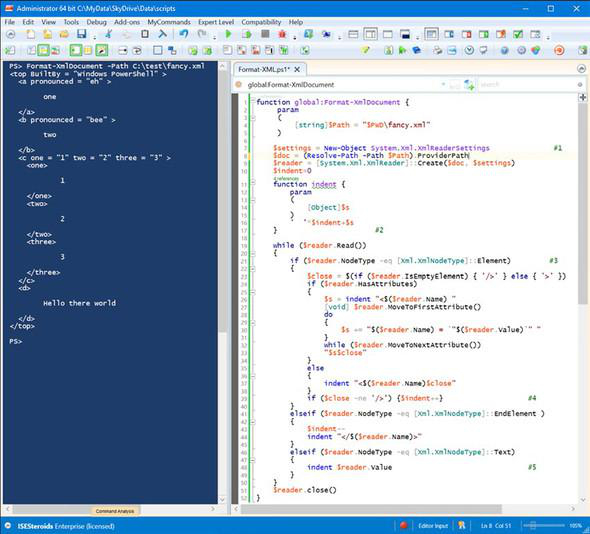
Format-XmlDocument is a complex function, so it’s worthwhile to take it one piece at a time. Let’s start with the basic function declaration, where it takes an optional argument that names a file. Next, you create the settings object 1 you need to pass in when you create the XML reader object. You also need to resolve the path to the document, because the XML reader object requires an absolute path. Now you can create the XmlReader object itself. The XML reader will stream through the document, reading only as much as it needs, as opposed to reading the entire document into memory.

You want to display the levels of the document indented, so you initialize an indent-level counter and a local function 2 to display the indented string. Now you read through all of the nodes in the document. You choose different behavior based on the type of the node. An element node 3 is the beginning of an XML element. If the element has attributes, then you add them to the string to display. You use the MoveToFirstAttribute() and MoveToNextAttribute() methods to move through the attributes. If there are no attributes, display the element name.

At each new element, increase 4 the indent level if it’s not an empty element tag. If it’s the end of an element, decrease the indent level and display the closing tag. If it’s a text element, display the value of the element 5. Finally, close the reader. You always want to close a handle received from a .NET method. It’ll eventually be discarded during garbage collection, but it’s possible to run out of handles before you run out of memory.

The function and its output are illustrated in [figure 16.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16fig05).

Figure 16.5: Format-XmlDocument displaying the test document in the ISE. Note that the ISE looks slightly different because ISEsteroids ([www.powertheshell.com/isesteroids/](http://www.powertheshell.com/isesteroids/)) is in use



The obvious next question is: How do you process XML documents on the pipeline?

#### Processing XML Documents in a Pipeline

Pipelining is one of the signature characteristics of shell environments in general and PowerShell in particular. Because the previous examples didn’t take advantage of this feature, we’ll look at how it can be applied. You’re going to write a function that scans all the PowerShell help files, both the text about topics and the XML files, for a specified pattern. The code for the function is shown in the following listing.

#### Listing 16.4: Search-Help function scans help files for a pattern

function Search-Help

{

param (

[Parameter(Mandatory)]

$pattern

)

Select-String -List $pattern -Path $PSHome\en-us\about\*.txt |

foreach {$\_.filename -replace '\..\*$'}

Get-ChildItem $PSHOME\en-us\\*dll-help.\*xml |

foreach { [xml] (Get-Content -ReadCount -1 -Path $\_) } |

foreach{$\_.helpitems.command} |

Where-Object {$\_.get\_Innertext() -match $pattern} |

foreach {$\_.details.name.trim()}

}

The Search-Help function takes one parameter to use as the pattern for which you’re searching. The $pattern parameter is set as mandatory so the user will be prompted if the parameter isn’t provided.

First, you search all the text files in the PowerShell installation directory and return one line for each matching file. Then you pipe this line into the ForEach-Object (or, more commonly, its alias foreach) to extract the base name of the file using the -replace operator and a regular expression. This operation will list the filenames in a form that you can type back into Get-Help.

Next, you get a list of the XML help files and turn each file into an XML object. You specify a read count of -1 so the whole file is read at once. You extract the command elements from the XML document and then see whether the text of the command contains the pattern you’re looking for. If it does, then you emit the name of the command, trimming off unnecessary spaces.

As an example of using the function, try this:

PS> Search-Help scriptblock

As well as being a handy way to search help, this function is a nice illustration of using the divide-and-conquer strategy when writing scripts in PowerShell. Each step in the pipeline brings you incrementally closer to the solution.

Another way to extract information from an XML document involves using XPath queries with Select-Xml.

#### XPath and Select-Xml

XML Path Language, also known as XPath, is a path-based pattern language, which means it’s like the collision of paths, wildcards, and regular expressions. It’s useful because it gives you a fast, concise way to select pieces of information from an XML document. An XPath expression can be used to extract nodes, content, or attributes from a document. It also allows calculations to be used in the expressions to get even greater flexibility. Things get a bit more complex because XML allows for multiple nodes with the same name and allows attributes on nodes.

Next, you’ll set up a test document and explore these more complex patterns. The following script fragment creates a string you’ll use for the examples. It’s a fragment of a bookstore inventory database. Each record in the database has the name of the author, the book title, and the number of books in stock. Save this string in a variable called $inventory, as shown here.

#### Listing 16.5: Creating the bookstore inventory

$inventory = @"

<bookstore>

<book genre="Autobiography">

<title>The Autobiography of Benjamin Franklin</title>

<author>

<first-name>Benjamin</first-name>

<last-name>Franklin</last-name>

</author>

<price>8.99</price>

<stock>3</stock>

</book>

<book genre="Novel">

<title>Moby Dick</title>

<author>

<first-name>Herman</first-name>

<last-name>Melville</last-name>

</author>

<price>11.99</price>

<stock>10</stock>

</book>

<book genre="Philosophy">

<title>Discourse on Method</title>

<author>

<first-name>Rene</first-name>

<last-name>Descartes</last-name>

</author>

<price>9.99</price>

<stock>1</stock>

</book>

<book genre="Computers">

<title>Windows PowerShell in Action</title>

<author>

<first-name>Bruce</first-name>

<last-name>Payette</last-name>

</author>

<price>39.99</price>

<stock>5</stock>

</book>

</bookstore>

"@

We’ll work through examples of using XPath with Select-Xml on the $inventory string from [listing 16.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16ex05). Let’s start with something simple—getting the bookstore node at the root of the document:

PS> Select-Xml -Content $inventory -XPath /bookstore

Node Path Pattern

---- ---- -------

bookstore InputStream /bookstore

Unfortunately, the output doesn’t look promising. The node object you’re after is mixed in with the context of the query: where the processed text came from and what the query was. To extract the node object, you have to reference it as a property:

PS> (Select-Xml -Content $inventory -XPath /bookstore).Node

book

----

{book, book, book, book}

This output shows that there are four child nodes under bookstore. Extend your query to get these child items in a similar manner to how you could get the contents of a directory in the file system:

PS> Select-Xml -Content $inventory -XPath /bookstore/book

Node Path Pattern

---- ---- -------

book InputStream /bookstore/book

book InputStream /bookstore/book

book InputStream /bookstore/book

book InputStream /bookstore/book

Here’s the nested-node issue again. Again you have to use the . operator to retrieve the content. For each of the nodes, you need to extract the Node property, so you apply the foreach cmdlet:

PS> Select-Xml -Content $inventory -XPath /bookstore/book |

foreach { $\_.node }

genre : Autobiography

title : The Autobiography of Benjamin Franklin

author : author

price : 8.99

stock : 3

<output truncated for brevity>

This time you see the properties of all four nodes. If you want to extract only the title nodes, add title to the end of the path:

PS> Select-Xml -Content $inventory -XPath /bookstore/book |

foreach { $\_.node.title }

The Autobiography of Benjamin Franklin

Moby Dick

Discourse on Method

Windows PowerShell in Action

At this point, using foreach all the time is getting tedious, so let’s define a filter to simplify this:

PS> filter node { $\_.node }

Now let’s look at more advanced examples. So far, you’ve returned the entire set of nodes, but when querying for information, you usually want to get part of that information. You can do this quite easily with the Where-Object cmdlet:

PS> Select-Xml -Content $inventory -XPath /bookstore/book |

node | where { [double] ($\_.price) -lt 10}

genre : Autobiography

title : The Autobiography of Benjamin Franklin

author : author

price : 8.99

stock : 3

genre : Philosophy

title : Discourse on Method

author : author

price : 9.99

stock : 1

This example retrieves all the books priced less than $10. XPath has built-in functionality that’s similar to the Where-Object cmdlet: predicate expressions. These expressions appear in the path surrounded by square brackets and can contain a simple logical expression. Nodes where the expression evaluates to true are returned. Here’s the previous example using a predicate expression instead of the Where-Object cmdlet:

PS> Select-Xml -Content $inventory -XPath '/bookstore/book[price < 10]' |

node

You get the same result in both cases. Notice that in the predicate expression you were able to reference price directly as opposed to [double] ($\_.price) the way you did in the Where-Object case. Because the expression is being executed by the XPath engine, it can make these optimizations, simplifying the reference to the price item and treating it as a number automatically.

In the previous example, the price item was a path relative to the current node. You can use .. to reference the parent node. Now write your expression so that it returns only the titles of the books whose price is less than $10:

PS> Select-Xml -Content $inventory `

-XPath '/bookstore/book/title[../price < 10]' |

node

#text

-----

The Autobiography of Benjamin Franklin

Discourse on Method

The path selects the title node but filters on the path ../price, which is a sibling to the title node.

As we discussed earlier, elements aren’t all that an XML document can contain. Another major item is the attribute. XPath allows an attribute to be referenced instead of an element by prefixing the name with @, as you see here:

PS> Select-Xml -Content $inventory -XPath '//@genre' | node

#text

-----

Autobiography

Novel

Philosophy

Computers

This example shows the genre attribute for each of the book nodes. You can also use attributes in predicate expressions in the path:

PS> Select-Xml -Content $inventory `

-XPath '//book[@genre = "Novel"]' |

node

genre : Novel

title : Moby Dick

author : author

price : 11.99

stock : 10

This example uses the @genre attribute in the node to return only books in the Novel genre. Note that, unlike the PowerShell relational operators, XPath operators are case-sensitive. If you specify novel for the genre instead of Novel, nothing is retrieved, whereas doing the same thing with the Where-Object cmdlet works fine.

#### Note

Remember that XPath is its own language and doesn’t necessarily behave the same way as the equivalent expression in PowerShell.

Now let’s do some processing on the data in the document instead of only retrieving the node. In this example, you’ll calculate the total value of the inventory, which is the sum of the product of multiplying the price node and the stock node:

PS> Select-Xml -Content $inventory -XPath '//book' | node |

foreach {[double] $\_.price \* $\_.stock } |

Measure-Object -Sum | foreach { $\_.sum }

356.81

This code uses XPath to extract the relevant nodes and then uses PowerShell to perform the calculations.

The examples in this section illustrate the basic mechanism for using XPath to extract data from documents. They’re far from comprehensive, though. There’s a lot more to learn about the details of the XPath language—the functions it supports, how to do calculations, and so forth—but this level of detail is probably not needed for most scenarios because PowerShell can do all of these things in a much more flexible way.

#### Rendering Objects as XML

Up to this point, you’ve been working with XML as objects. Now you’re going to switch it around and render objects into XML using cmdlets. PowerShell provides two cmdlets for rendering objects as XML, each with slightly different purposes. The ConvertTo-Xml cmdlet renders objects with relatively simple but verbose format. This cmdlet is useful for interoperating between PowerShell and other environments. Conversions using Export-Clixml are much more complex but also more compact and are intended for efficiently passing data between instances of PowerShell.

#### Note

ConvertTo-XmL doesn’t automatically create an output file, but Export -CliXml does.

We’ll start with the simpler of the cmdlets: ConvertTo-Xml which takes an object as an argument or (more commonly) as pipeline input and generates an XML document from it. Let’s use it to produce XML from a list of Windows services. You’ll get the list using the Get-Service cmdlet, but you’ll limit the number of services you’ll work with to three for brevity’s sake:

PS> $doc = Get-Service | select -First 3 | ConvertTo-Xml

PS> $doc

xml Objects

--- -------

version="1.0" encoding="utf-8" Objects

The collection of objects is rendered into an XML document with the top node Object, which, in turn, contains a collection of Object elements as shown:

PS> $doc.Objects.Object

Type Property

---- --------

System.ServiceProcess.ServiceCo... {Name, RequiredServices, CanPau...

System.ServiceProcess.ServiceCo... {Name, RequiredServices, CanPau...

System.ServiceProcess.ServiceCo... {Name, RequiredServices, CanPau...

Here you see that each Object element has the type and properties of the source object included in the output document. But this representation doesn’t show the document format effectively, so use the -As parameter to display the document as a single string:

PS> Get-Service | select -First 1 | ConvertTo-Xml -As String

<?xml version="1.0" encoding="utf-8"?>

<Objects>

<Object Type="System.ServiceProcess.ServiceController">

<Property Name="Name" Type="System.String">AJRouter</Property>

<Property Name="RequiredServices" Type="System.ServiceProcess.ServiceController[]" />

<Property Name="CanPauseAndContinue" Type="System.Boolean">False</Property>

<Property Name="CanShutdown" Type="System.Boolean">False</Property>

<Property Name="CanStop" Type="System.Boolean">False</Property>

<Property Name="DisplayName" Type="System.String">AllJoyn Router Service</Property>

<Property Name="DependentServices" Type="System.ServiceProcess.ServiceController[]" />

<Property Name="MachineName" Type="System.String">.</Property>

<Property Name="ServiceName" Type="System.String">AJRouter</Property>

<Property Name="ServicesDependedOn" Type="System.ServiceProcess.ServiceController[]" />

<Property Name="ServiceHandle" Type="SafeServiceHandle">SafeServiceHandle</Property>

<Property Name="Status" Type="System.ServiceProcess.ServiceControllerStatus">Stopped</Property>

<Property Name="ServiceType" Type="System.ServiceProcess.ServiceType">Win32ShareProcess</Property>

<Property Name="StartType" Type="System.ServiceProcess.ServiceStartMode">Manual</Property>

<Property Name="Site" Type="System.ComponentModel.ISite" />

<Property Name="Container" Type="System.ComponentModel.IContainer" />

</Object>

</Objects>

Now the structure of the saved data is much clearer. The type name of the original object is included as an attribute on the Object tab. The child elements of Object are a collection of Property objects with the property name and type as attributes and the value as the element content.

One thing we didn’t mention was the serialization depth. The default depth is 2. You see this in the RequiredServices property, whose content is two additional nested properties. You can override the default depth using the -Depth parameter on the cmdlet.

#### Note

You might be tempted to set the depth to a larger value to preserve more information, but be aware that the size of the document can explode with deep nesting. For example, saving the process table with the default depth of 2 produces a 700 KB file, which is already quite large. Increasing the depth to 3 explodes the file to 7 MB—a tenfold increase in size!

The other parameter on the cmdlet that we haven’t talked about is -NoTypeInformation. When you specify this parameter, no type information is included in the generated document:

PS> Get-Service | select -First 1 |

ConvertTo-Xml -As String -NoTypeInformation

<?xml version="1.0" encoding="utf-8"?>

<Objects>

<Object>

<Property Name="Name">AJRouter</Property>

<Property Name="RequiredServices" />

<Property Name="CanPauseAndContinue">False</Property>

<Property Name="CanShutdown">False</Property>

<Property Name="CanStop">False</Property>

<Property Name="DisplayName">AllJoyn Router Service</Property>

<Property Name="DependentServices" />

<Property Name="MachineName">.</Property>

<Property Name="ServiceName">AJRouter</Property>

<Property Name="ServicesDependedOn" />

<Property Name="ServiceHandle">SafeServiceHandle</Property>

<Property Name="Status">Stopped</Property>

<Property Name="ServiceType">Win32ShareProcess</Property>

<Property Name="StartType">Manual</Property>

<Property Name="Site" />

<Property Name="Container" />

</Object>

</Objects>

This simplifies the output even further. It makes sense if the target consumer for the generated document isn’t a .NET-based application and therefore won’t be able to do much with the type names.

The ConvertTo-XML cmdlet is useful for interoperation with non-PowerShell applications, but for PowerShell-to-PowerShell communication, too much information is lost. For the PowerShell-to-PowerShell scenario, a much better solution is to use the Export-Clixml and Import-Clixml cmdlets, which provide a way to save and restore collections of objects from the PowerShell environment with higher fidelity (less data loss) than the ConvertTo-Xml cmdlet.

The encoding the \*-Clixml cmdlets use is what PowerShell remoting uses to send objects between hosts. To recap our discussion, we mentioned that only a small set of types serialize with fidelity and that other types are shredded into property bags. With the \*-Clixml cmdlets, you can see what the encoding looks like. Let’s try this out. First, create a collection of objects: a hashtable, a string, and some numbers. Then serialize them to a file using the Export-Clixml cmdlet:

PS> $data = @{a=1;b=2;c=3},"Hi there", 3.5

PS> $data | Export-Clixml -Path C:\test\out.xml

Let’s see what the file looks like:

PS> Get-Content -Path C:\test\out.xml

<Objs Version="1.1.0.1" xmlns="http://schemas.microsoft.com/powershell/2004/04">

<Obj RefId="0">

<TN RefId="0">

<T>System.Collections.Hashtable</T>

<T>System.Object</T>

</TN>

<DCT>

<En>

<S N="Key">c</S>

<I32 N="Value">3</I32>

</En>

<En>

<S N="Key">b</S>

<I32 N="Value">2</I32>

</En>

<En>

<S N="Key">a</S>

<I32 N="Value">1</I32>

</En>

</DCT>

</Obj>

<S>Hi there</S>

<Db>3.5</Db>

</Objs>

You can use Import-Clixml to re-create the data. To show that the resultant object is identical to the original test with Compare-Object, do this:

PS> Compare-Object -ReferenceObject $data `

-DifferenceObject (Import-Clixml -Path C:\test\out.xml) `

-IncludeEqual

InputObject SideIndicator

----------- -------------

{c, b, a} ==

Hi there ==

3.5 ==

These cmdlets provide a simple way to save and restore collections of objects, but they have limitations. They can load and save only a fixed number of primitive types. Any other type is “shredded,” which means it’s broken apart into a property bag composed of these primitive types. This allows any type to be serialized but with some loss of fidelity. Objects can’t be restored to exactly the same type they were originally. This approach is necessary because there can be an infinite number of object types, not all of which may be available when the file is read back. Sometimes you don’t have the original type definition. Other times there’s no way to re-create the original object, even with the type information, because the type doesn’t support this operation. By restricting the set of types that are serialized with fidelity, the Clixml format can always recover objects regardless of the availability of the original type information.

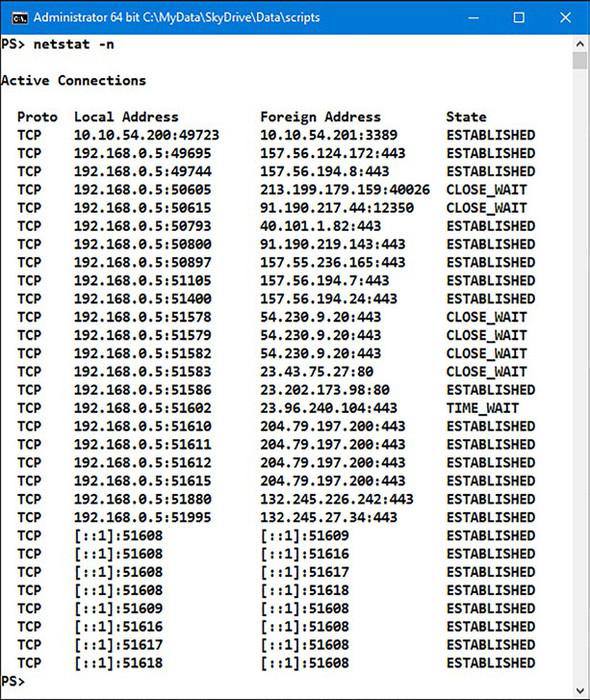
There’s also another limitation on how objects are serialized. An object has properties. Those properties are also objects that have their own properties, and so on. This chain of properties that have properties is called the serialization depth. For some of the complex objects in the system, such as the Process object, serializing through all the levels of the object results in a huge XML file. To constrain this, the serializer traverses only to a certain depth. The default depth is 2. You can override this default either on the command line using the -Depth parameter or by placing a <SerializationDepth> element in the type’s description file. If you look at $PSHome/types.ps1xml, you can see some examples of where this has been done.

So far, we’ve discussed manipulating files, strings, and text data. Now we need to discuss how to convert text data into objects.

### 16.2.4 Converting Text Output to Objects

One of the first things everyone learns about PowerShell is it works with objects. You can execute legacy command-line applications in the PowerShell console but you get text output, as shown in [figure 16.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16fig06).

Figure 16.6: Output of the netstat.exe legacy application



Converting the output from text to objects is possible, but it requires a lot of work and the results can be quite fragile if the application’s output changes. PowerShell v5 has a cmdlet—ConvertFrom-String—that makes these conversions much simpler. Referring to [figure 16.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16fig06) as you progress through this section will clarify the code.

If you start by passing the output of netstat through Convert-String, you’ll discover the first issue:

PS> netstat -n | ConvertFrom-String | Select-Object -First 5

P1 P2

-- --

Active Connections

Proto

TCP

TCP

TCP

The header line Active Connections is split to create two properties, P1 and P2. The default delimiter is whitespace. The contents of the Proto (protocol) field are assigned to P2 and all other data is dropped. Not quite what you wanted!

Let’s skip the first three lines:

PS> netstat -n | select -Skip 3 | ConvertFrom-String

Each line of netstat output looks something like this:

P1 :

P2 : TCP

P3 : 10.10.54.200:49723

P4 : 10.10.54.201:3389

P5 : ESTABLISHED

The first set of outputs is the field headers. Notice that the properties are given consecutive names: P1, P2, P3, and so on. The next step is to discard the field header row and assign useful names to the properties:

PS> netstat -n | select -Skip 4 |

ConvertFrom-String -PropertyNames Protocol,

LocalAddress, ForeignAddress, State

The output is a number of objects of this form:

Protocol :

LocalAddress : TCP

ForeignAddress : 10.10.54.200:49723

State : 10.10.54.201:3389

P5 : ESTABLISHED

The properties are assigned incorrectly because there are multiple whitespaces at the beginning of each line. One way to deal with that is to assign a dummy property:

PS> netstat -n | select -Skip 4 |

ConvertFrom-String -PropertyNames Blank, Protocol,

LocalAddress, ForeignAddress, State

The output looks like this:

Blank :

Protocol : TCP

LocalAddress : 10.10.54.200:49723

ForeignAddress : 10.10.54.201:3389

State : ESTABLISHED

You should filter out the Blank property because it’s not required:

PS> netstat -n | select -Skip 4 |

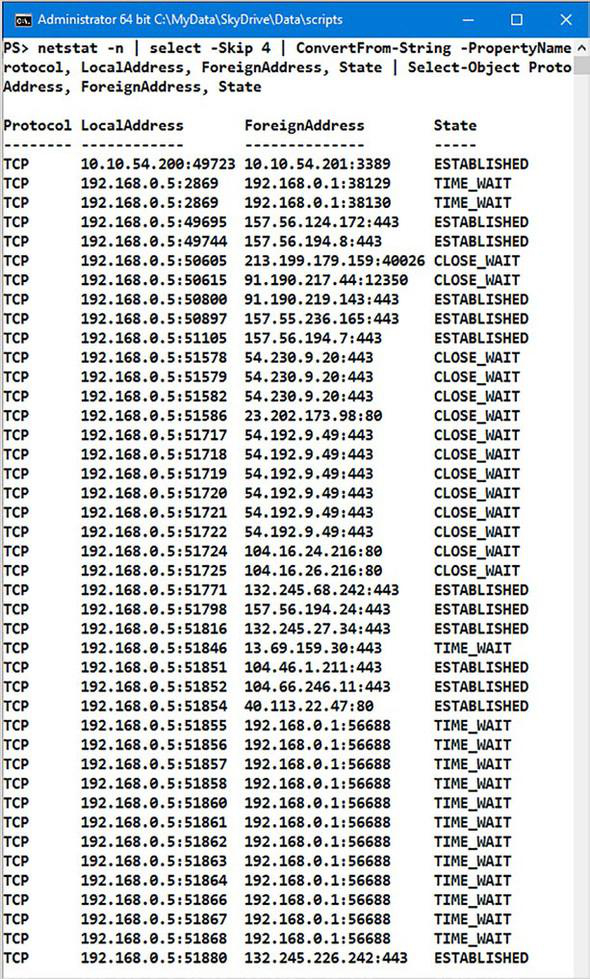
ConvertFrom-String -PropertyNames Blank, Protocol,

LocalAddress, ForeignAddress, State |

Select-Object Protocol, LocalAddress, ForeignAddress, State

This will produce the output shown in [figure 16.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16fig07).

Figure 16.7: Result of processing netstat output with ConvertFrom-String



[Figures 16.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16fig06) and [16.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-162-127.xhtml#ch16fig07) look similar. The important point is now that you can convert the netstat output to objects, you can apply standard PowerShell techniques to filter the data. For example:

PS> $nso = netstat -n | select -Skip 4 |

ConvertFrom-String -PropertyNames Blank, Protocol, LocalAddress, ForeignAddress, State |

Select-Object Protocol, LocalAddress, ForeignAddress, State

PS> $nso.where({$\_.State -eq 'ESTABLISHED'})

PS> $nso | sort State

PS> $nso | where LocalAddress -like '192.168.0.5\*' | sort ForeignAddress

That closes our look at files and working with text data. It’s time to turn your attention to the older COM object model and how you can work with it in PowerShell.

## 16.3 Accessing COM Objects

COM is an interface specification describing how to write libraries that can be used from multiple languages or environments. Prior to technologies like COM, each programming language required its own set of libraries. The COM specification allowed the creation of libraries of components that could be accessed from multiple languages. But beyond sharing library code, COM allowed running applications to expose automation interfaces that external programs could use to remotely control them. In this section, we’ll introduce COM and show you how to leverage COM classes using PowerShell. COM provides easy (and in some cases trivial) access to many Windows features. We’ll work through a number of examples in a variety of application scenarios, and we’ll complete our COM coverage by examining some of the issues and limitations the PowerShell scripter may encounter.

### Creating COM Objects

The first thing you need to know if you want to work with COM (or any other object system for that matter) is how to create instances of COM objects. As with .NET objects, you use the New-Object cmdlet, but for COM objects you have to specify the -ComObject parameter:

PS> $word = New-Object -ComObject 'Word.application'

Unlike .NET objects, COM doesn’t have a way to pass arguments to the object’s constructor, making it hard to initialize the object. As a workaround for this, in PowerShell v2 (and later) you use the -Property parameter on New-Object to initialize properties on the constructed object before returning it.

Unique to the COM parameter set is the -Strict switch. This switch tells the cmdlet to generate an error if a .NET/COM Interop library is loaded. In [chapter 2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-2-16.xhtml#ch02), we talked about how the PowerShell type system uses adaptation to give the user a consistent experience with different kinds of objects. COM objects are one of these adapted types, but the way the PowerShell adapter works is affected by the presence or absence of a COM Interop library.

In effect, this Interop library is .NET’s own adaptation layer for COM. The net effect is that the PowerShell COM adapter will project a different view of a COM object if an Interop library is loaded versus when there’s not one. This becomes a problem because, for any given COM class on any given machine, there may or may not be an Interop library, so you may or may not get the doubly adapted COM object. If you want to be able to write scripts that behave consistently everywhere, you need a way to control how the adaptation is done. The -Strict parameter allows you to detect this when an Interop library is loaded. Once you know what’s happening, you can decide whether you want to fail or continue but along a different code path. This kind of portability issue is something to keep in mind when you’re writing a script using COM that you plan to deploy on other machines. But for now, let’s move on to our next topic and see how to find out which COM classes are available.

### Identifying and Locating COM Classes

Officially, all COM classes are identified by a globally unique ID (GUID). This isn’t a particularly friendly way to identify, well, anything. As far as PowerShell is concerned, COM objects are identified by a much more usable name called the ProgID. This is a string alias that’s provided when the class is registered on the system. Using the ProgID is the most human-friendly way of identifying the object. By convention, the ProgID has the form

<Program>.<Component>.<Version>

which (at least according to the MSDN documentation) should be fewer than 39 characters in length.

#### Note

Although this format is the recommended way to create a ProgID, there’s no real way to enforce it, resulting in some interesting interpretations of what each of the elements means. Generally, it seems in practice that <Program> is the application suite, toolset, or vendor that installed it; <component> is the COM class name; and the version number is normally not used in calls, though it may exist in even a multipart form.

COM objects are registered in (where else?) the Registry. This means that you can use the Registry provider to search for ProgIDs from PowerShell.

#### Listing 16.10: Discovering ProgIds

function Get-ProgId

{

param (

$filter = '.'

)

Get-ChildItem -Path 'REGISTRY::HKey\_Classes\_Root\clsid\\*\progid' |

foreach {if ($\_.name -match '\\ProgID$') { $\_.GetValue('') }} |

Where-Object {$\_ -match $filter}

}

Using the function with the default search filter

PS> Get-ProgId | Sort-Object

will return all available ProgIds. If you want to restrict the search—for instance, for the ProgId for Internet Explorer—use the filter:

PS> Get-ProgId -filter internet

#### Note

The CIM class Win32\_ProgIDSpecification will return some but not all ProgIds. The safest option is to use the function in [listing 16.10](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-163-128.xhtml#ch16ex10).

As with everything else in PowerShell, examples save thousands of words, so we’ll show you examples of working with COM objects, starting with how to automate some Windows basic features.

### Automating Windows with COM

The Shell.Application class provides access to Windows Explorer and its capabilities. It allows automation of many shell tasks, like opening file browser windows, launching documents or the help system, finding printers, computers, or files, and so on. The first thing you need to do is create an instance of this class:

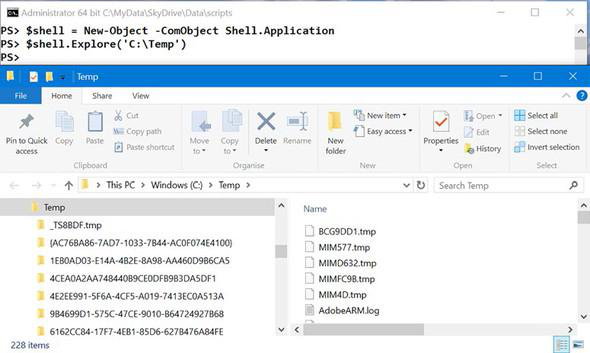
PS> $shell = New-Object -ComObject Shell.Application

As always in PowerShell, COM objects, like any other object type, can be examined using Get-Member. It’s worth exploring the available methods; for instance, the Explore() method, which will launch an Explorer window on the path specified:

PS> $shell.Explore('C:\Temp\')

At this point, you should see something like [figure 16.8](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-163-128.xhtml#ch16fig08). This method call opened an Explorer window in the Temp directory of the C: drive.

Figure 16.8: Launching Windows Explorer on C:\Temp



Here’s a handy function for laptop users who move around a lot. Many laptops have docking stations that allow you to easily connect multiple peripherals. This is great except that you need to undock the laptop before heading to a meeting. Depending on the laptop, this can be annoying, so here’s a quick one-line function to undock a laptop:

PS> function eject { ( New-Object -ComObject Shell.Application ).EjectPC()}

This function gets an instance of the Shell.Application object and then calls the EjectPC() method to undock the laptop.

The Windows() method on Shell.Application allows you to get a list of the Explorer and Internet Explorer windows that are currently open:

PS> $shell.Windows() | select Name, LocationURL

or you can index directly into the collection of Windows:

PS> $shell.Windows()[0] | select Name, LocationURL

Name LocationURL

---- -----------

File Explorer file:///C:/Temp

#### PowerShell 5.0 changes

Prior to PowerShell v5 you’d have received an error when trying to index into the collection of Windows:

PS> $shell.Windows()[0]

Unable to index into an object of type System.\_\_ComObject.

At line:1 char:18

+ $shell.Windows()[0 <<<< ]

The error occurred because the PowerShell interpreter didn’t know how to index these collections, because they consisted of the COM object inside a .NET wrapper, which was then adapted by PowerShell. You had to use this syntax:

PS> $shell.Windows().Item(0)

A number of changes were made to the way COM objects are processed during the development of PowerShell v5, including being able to understand the wrappers and so index into the collection.

The other big improvement was in processing speed. Utilizing COM objects could be glacially slow, in particular the Excel objects when writing to a spreadsheet. PowerShell v5 shows a significant increase in processing speed for COM objects.

Closing a window requires the Quit() method:

PS> $shell.Windows()[3].Quit()

You can even close a set of windows in one pass:

PS> $shell.Windows() | where LocationURL -match 'amazon' |

foreach {$\_.Quit()}

There are many other methods for you to explore using the Shell.Application class. But for now, we’ll turn our attention to Microsoft Word.

### Using Microsoft Word for Spell Checking

Wouldn’t it be great if every environment you worked in had spell checking, like word processors do? With PowerShell and COM, you can get at least part of the way there. You’re going to write a script that will use Microsoft Word to spell check the contents of the clipboard and then paste them back. You’ll call this script Get-Spelling.ps1. Try this in both the PowerShell console and the ISE.

Let’s see how it’s used. First, put some text, with errors, on the clipboard:

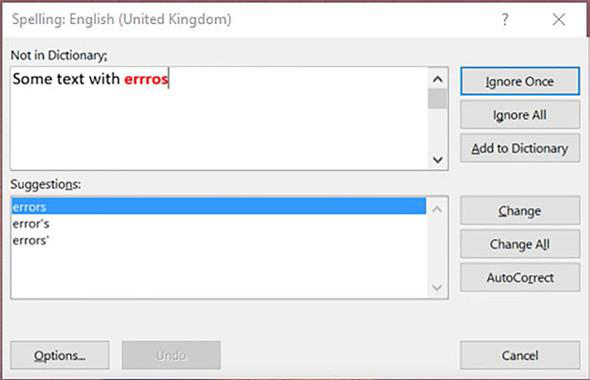
PS> Set-Clipboard -Value 'Some text with errros'

Now run the function from [listing 16.11](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-163-128.xhtml#ch16ex11):

PS> Test-Spelling

You’ll see the Word Spelling dialog box pop up, as shown in [figure 16.9](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-163-128.xhtml#ch16fig09).

Figure 16.9: The Microsoft Word spell checker launched by the Test-Spelling function shows the misspelled text that was copied from the clipboard



You need to go through all the spelling errors and fix them as appropriate. Once all the errors are fixed, the dialog box will disappear, and the pop-up box will be displayed, indicating that the revised text is available in the clipboard.

#### Note

The \*-Clipboard cmdlets were introduced in PowerShell v5.

You can view the changed text:

PS> Get-Clipboard

Some text with errors

and you’re finished. The text is correctly spelled. Now that you know how to use this script, let’s look at the Test-Spelling code.

#### Listing 16.11: The Test-Spelling function

function Test-Spelling {

$wshell = New-Object -ComObject WScript.Shell

$word = New-Object -ComObject Word.Application

$word.Visible = $false

$doc = $word.Documents.Add()

$word.Selection.Paste()

if ($word.ActiveDocument.SpellingErrors.Count -gt 0)

{

$word.ActiveDocument.CheckSpelling()

$word.Visible = $false

$word.Selection.WholeStory()

$word.Selection.Copy()

$wshell.PopUp( 'The spell check is complete, ' +

'the clipboard holds the corrected text.' )

}

else

{

[void] $wshell.Popup('No Spelling Errors were detected.')

}

$x = [ref] 0

$word.ActiveDocument.Close($x)

$word.Quit()

}

The first thing you do is create the object instances you’re going to use. You need an instance of WScript.Shell to pop up a message box and the Word.Application object for the bulk of the work. Once you have the Word.Application object, you make the Word window invisible and then add an empty document to hold the text you want to spell-check.

Next, you copy the contents from the clipboard to the Word document you created and see if you need to spell check the text. If you do, you present the Spelling dialog box. When the spell check is complete, you select all the text and copy it back to the clipboard so you can paste it into the original document and inform the user that the corrected text is available. If there were no spelling errors, you’d display a message box confirming this. The last step is to discard the document you created and close the application. With this script, you can add spell-checking capabilities to any application that lets you select and copy text.

#### Note

Obviously, if Microsoft Word isn’t your word processor of choice, it should be simple to modify the script to work with any word processor that exports a similar automation model.

Using COM in PowerShell lets you automate applications, but there are also issues with COM support, which we’ll cover in the next section.

### Issues with COM

Support for COM in PowerShell is good but not perfect. In part, this is because PowerShell depends on .NET, and .NET’s support for COM is also not perfect. In this section, we’ll explore a few problems that you may run into when using COM from PowerShell, including more information on the Interop assembly issue.

One problem that arises is that some COM objects are available only to 32-bit applications. On 64-bit systems, the 64-bit PowerShell binaries are run by default, so if you need to use a 32-bit–only COM object, you’ll have to explicitly start the 32-bit version of PowerShell. This can also be an issue when using remoting because the default remoting configuration on 64-bit systems is 64-bit as well. To remotely run a script that requires a 32-bit COM object, you’ll have to connect to the 32-bit configuration on the remote machine, regardless of whether the local system is 32- or 64-bit.

Another thing that can potentially cause problems has to do with the way the COM object has been wrapped or adapted. There are three possible categories of COM objects you may encounter: a COM object that has a .NET Interop library, a COM object that has a type library (commonly called a typelib) but no Interop assembly, and a COM object that has neither.

In the first category, you get a COM object that has been wrapped in a .NET Interop wrapper. This wrapper may introduce changes in the object’s interface or behavior that affect how you work with that object compared to the raw COM object. For this reason, the New-Object cmdlet’s ComObject parameter set has an additional parameter, -Strict, that causes a nonterminating error to be written if an Interop assembly is loaded. Let’s look at examples. Start by creating an instance of the Word.Application object you used earlier:

PS> $word = New-Object -ComObject Word.Application

Now try it again but with the -Strict parameter:

PS> $word = New-Object -ComObject Word.Application -Strict

New-Object : The object written to the pipeline is an instance of the type

"Microsoft.Office.Interop.Word.ApplicationClass" from the component's

primary interoperability assembly. If this type exposes different members

than the IDispatch members, scripts that are written to work with this

object might not work if the primary interoperability assembly is not

installed.

At line:1 char:9

+ $word = New-Object -ComObject Word.Application -Strict

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidArgument: (Microsoft.Offic...pplication

Class:ApplicationClass) [New-Object], PSArgument

Exception

+ FullyQualifiedErrorId : ComInteropLoaded,Microsoft.PowerShell.Commands.NewObjectCommand

You get a detailed error message explaining that the object that was loaded is a wrapped object. Note that this is a nonterminating error message, so the object is still returned and execution proceeds. Here’s how to use this feature to write a script that can adapt its behavior appropriately.

First, you don’t want the error message to appear in the output of your script, so redirect it to $null. But even when you do this, the $? variable, which indicates whether the last command executed was successful, is still set to $false so you know that an error occurred:

PS> $word = New-Object -ComObject Word.Application -Strict 2> $null

PS> $?

False

A script should check this variable and take an alternate action for the wrapped and nonwrapped cases. Investigating further, let’s see what was returned by the call to New-Object:

PS> $word.GetType().Fullname

Microsoft.Office.Interop.Word.ApplicationClass

The output shows that the object is an instance of the Interop assembly mentioned earlier.

Next, look at an object for which there’s no Interop assembly and see how that behaves differently. Create an instance of the Shell.Application class you worked with earlier:

PS> $shell = New-Object -ComObject Shell.Application

PS> $shell | Get-Member

TypeName: System.\_\_ComObject#{efd84b2d-4bcf-4298-be25-eb

542a59fbda}

Name MemberType Definition

---- ---------- ----------

AddToRecent Method void AddToRecent (Varian...

BrowseForFolder Method Folder BrowseForFolder (...

:

In this situation, you see that the type of the object is System.\_\_ComObject followed by the GUID of the registered type library. This type library allows you to see the members on the object but doesn’t affect the object’s behavior.

There is another type of object you need to consider: those created using CIM.

## 16.4 Using CIM

CIM is an industry standard (a set of related standards) created by Microsoft, HP, IBM, and many other computer companies with the goal of defining a common set of management abstractions. By creating interoperable common models for managed elements like services, processes, or CPUs, you can start to build management tools and processes that can be applied universally. WMI is Microsoft’s original implementation of CIM.

#### Note

In this section we’ll show you how the CIM cmdlets work. If you want to dig deeper into CIM (WMI), you should read PowerShell and WMI by Richard Siddaway (Manning Publications, 2012), which covers the WMI and CIM cmdlets and how to use them to administer Windows systems.

There are standard ways of wrapping bits of management data in a well-defined package so you can work with this data across different vendors and environments in a consistent way; these are the standard or base CIM classes. To support environment-specific extensions, CIM also allows vendors to create derived classes of the CIM base classes that can surface nonstandard features as a set of extensions while still preserving the common base characteristics of the model. The goal of all this is to make it easier to create system administration tools (and, by corollary, system administrators) that can work effectively in heterogeneous environments. In the next section, we’ll look at how the CIM/WMI infrastructure facilitates these goals.

#### CIM and WMI

One point of confusion that needs to be cleared up immediately is the difference between CIM and WMI. The short and simple answer is that there isn’t any difference.

CIM is a standard created by the Distributed Management Task Force (DMTF) to provide a common definition of management information across computers, networks, applications, and services; see [www.dmtf.org/standards/cim](http://www.dmtf.org/standards/cim). The DMTF defines CIM like this:

“CIM provides a common definition of management information for systems, networks, applications, and services, and allows for vendor extensions. CIM’s common definitions enable vendors to exchange semantically rich management information between systems throughout the network.”

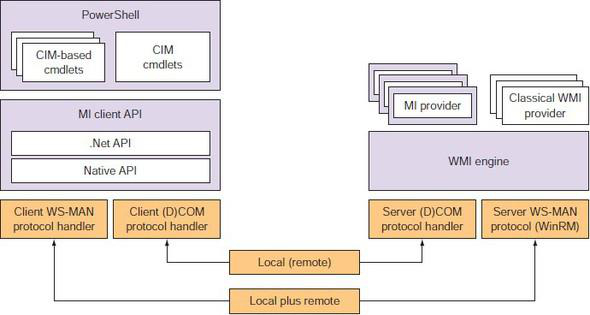
WMI is the name Microsoft gave its original implementation of the CIM standard.

The new API and cmdlets introduced with PowerShell v3 use the CIM prefix to distinguish them from the WMI cmdlets introduced in PowerShell v1 and v2. The WMI cmdlets use DCOM (Distributed Component Object Model) to connect to remote machines. The newer CIM cmdlets use WS-MAN for remote connectivity in a similar way to PowerShell remoting.

PowerShell has supported CIM since version 1.0; in fact, Get-WmiObject was the only cmdlet in the original PowerShell version that had the capability to access remote machines. The level of CIM support in PowerShell has increased with subsequent versions. This section will explain what CIM is, how to access it from PowerShell, and what you can do with CIM once you have this access. You’ll work through a number of examples to see how things work, exploring the sorts of tasks that can be accomplished.

PowerShell v3 introduced a new API and cmdlets for working with WMI. Usually referred to as the CIM cmdlets, they have the ability to create and use connections to remote machines in a similar manner to PowerShell remoting. We’ll concentrate on the CIM cmdlets in this section rather than the older WMI cmdlets. The CIM cmdlets support both the original WMI providers and the newer APIs, as shown in [figure 16.10](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-164-129.xhtml#ch16fig10).

Figure 16.10: CIM cmdlet support for original WMI providers and modern API



### 16.4.1 The CIM Cmdlets

The CimCmdlets module supplies a number of cmdlets for working with CIM, as shown in [table 16.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-164-129.xhtml#ch16table02). The equivalent WMI cmdlet is provided where applicable.

| **Table 16.2: The CIM cmdlets and their purpose compared with the WMI cmdlets** | | |
| --- | --- | --- |
| **CIM cmdlet** | **Purpose** | **Equivalent WMI cmdlet** |
| Get-CimClass | Retrieves CIM class structure | None |
| Get-CimInstance | Retrieves objects from CIM | Get-WmiObject |
| Get-CimAssociatedInstance | Retrieves associated CIM instances | None |
| Invoke-CimMethod | Invokes a method on a CIM class | Invoke-WmiMethod |
| New-CimInstance | Creates a new instance of a CIM class | None |
| Register-CimIndicationEvent | Subscribes to events surfaced through CIM | Register-WmiEvent |
| Remove-CimInstance | Removes a CIM instance from the repository | Remove-WmiObject |
| Set-CimInstance | Sets the properties of a CIM instance | SetWmiInstance |

#### Finding CIM Classes

Before you can use Get-CimInstance to retrieve data, you need to know which class to use. CIM is a self-describing technology, which means it provides ways for a client application to ask the object manager on the target system what’s available. Get-CimClass leverages these mechanisms. For example, to see all of the classes with BIOS in their name, use this:

PS> Get-CimClass -ClassName \*bios\*

The output shows each of the available class names along with the methods and properties defined by those classes. Let’s look at a specific class:

PS> Get-CimClass -ClassName Win32\_Bios

NameSpace: ROOT/cimv2

CimClassName CimClassMethods CimClassProperties

------------ --------------- ------------------

Win32\_BIOS {} {Caption, Description ...}

As you saw with .NET, the amount of information returned from the commands is frequently enough for your purposes, but all the standard classes that Microsoft includes with Windows are well documented on MSDN: <http://mng.bz/43Da>.

This documentation includes many examples showing how to use classes. Many of the examples are written in VBScript, although this is becoming less true over time. CIM classes are arranged in namespaces. In the previous output, you saw

NameSpace: ROOT\cimv2

indicating that the classes listed were located in this namespace. Because this is PowerShell’s default namespace, you haven’t needed to use the -Namespace parameter yet. All classes are identified by a path of this form:

\\<computer>\<namespace>\<namespace>:<class>

#### Note

The vast majority of CIM classes that you’ll use on a regular basis live in the root\cimv2 namespace. PowerShell treats this as the default namespace, which is why you don’t force the use of the -Namespace parameter. Because the -ClassName parameter is positional, many CIM commands can be written as Get-CimInstance Win32\_Bios.

Namespaces can contain nested namespaces and classes. The set of CIM namespaces and classes available on a machine depends on what’s installed on that machine (both applications and operating system).

#### Selecting CIM Instances

[Table 16.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-164-129.xhtml#ch16table02) shows that Get-CimInstance (equivalent to the older Get-WmiObject) is used to retrieve data from the CIM repository. If you’re dealing with the default namespace, the cmdlet is used like this:

PS> Get-CimInstance -ClassName Win32\_Bios

SMBIOSBIOSVersion : 2.05.0250

Manufacturer : American Megatrends Inc.

Name : 2.05.0250

SerialNumber : 036685734653

Version : OEMA - 1072009

To run this same command against a remote computer, you have to add the -ComputerName parameter to the command. You get this:

PS> Get-CimInstance -ClassName Win32\_Bios -ComputerName W510W16

SMBIOSBIOSVersion : 6NET61WW (1.24 )

Manufacturer : LENOVO

Name : Ver 1.00PARTTBLX

SerialNumber : R81BG3K

Version : LENOVO - 1240

PSComputerName : W510W16

#### CIM cmdlet connectivity

The CIM cmdlets use different protocols to connect to systems depending on the scenario:

* Local machine (no use of -ComputerName) = COM
* Local machine using -ComputerName = WS-MAN
* Remote machine using -ComputerName = WS-MAN
* Remote machine using CIM session = WS-MAN
* Remote machine using CIM session using DCOM protocol = DCOM

If you want to restrict the data returned from a CIM call, the most efficient way is to use the -Filter parameter:

PS> Get-CimInstance -ClassName Win32\_NetworkAdapterConfiguration `

-ComputerName W510W16 -Filter "DHCPEnabled = $true"

This is the equivalent of using a WMI Query Language (WQL) query:

PS> Get-CimInstance -ComputerName W510W16 `

-Query "SELECT \* FROM Win32\_NetworkAdapterConfiguration

WHERE DHCPEnabled = $true"

The -Filter parameter takes the part of the WQL query after the WHERE keyword.

Like .NET and COM objects, WMI objects have methods. You’ll see how to invoke these methods in the next section.

#### Invoking CIM Methods

CIM classes can have both static or class members and object or instance members. Static methods are the easier types to call because you only need the class name, method name, and arguments to call. Instance methods are more complex because you need to specify additional information to identify which instance of the target class to invoke the method on.

As our test case for static methods, let’s use the static Create() method on the Win32\_Process class to create an instance of (start) a process—in this case, calc.exe. The command to do that looks like this:

PS> Invoke-CimMethod -ClassName Win32\_Process -MethodName Create `

-Arguments @{CommandLine = 'calc.exe'}

ProcessId ReturnValue PSComputerName

--------- ----------- --------------

3620 0

If the method call is successful, then the ReturnValue will be 0, indicating success. Any other value indicates failure and the error.The ProcessID will contain the process ID or handle of the new process. The arguments for the method are supplied as a hashtable. If you have multiple arguments, separate them with a semicolon (;):

PS> Invoke-CimMethod -ClassName Win32\_Process -MethodName Create `

-Arguments @{CommandLine = 'notepad.exe'; CurrentDirectory = 'C:\test'}

The easiest way to invoke an instance method is to get the instance and pass it to Invoke-CimMethod:

PS> Get-CimInstance -ClassName Win32\_Process -Filter "Name='calculator.exe'"

| Invoke-CimMethod -MethodName Terminate

A filter identifies the individual process, which is then passed to Invoke-CimMethod, and the Terminate() method is called. The process is shut down.

#### Deleting CIM Instances

The Remove-CimInstance cmdlet can be used to delete instances. You could use this instead of calling the Terminate() method:

PS> Get-CimInstance -ClassName Win32\_Process -Filter "Name='calculator.exe'" |

Remove-CimInstance

The final action you may need to take on a CIM instance is to modify its properties.

#### Modifying CIM Instances

Many of the properties on CIM classes are read-only; you can’t alter them. You can check if an individual property is read-only like this:

PS> (Get-CimClass -ClassName Win32\_OperatingSystem).

CimClassProperties['Manufacturer']

Name : Manufacturer

Value :

CimType : String

Flags : Property, ReadOnly, NullValue

Qualifiers : {MappingStrings, read}

ReferenceClassName :

If you want to see all the read-only properties, use this:

PS> Get-CimClass -ClassName Win32\_OperatingSystem |

select -ExpandProperty CimClassProperties |

where Flags -like '\*ReadOnly\*' |

select Name, CimType

Conversely, if you want to see the properties, you can change the -like operator to -notlike in the previous code. This shows that on the Win32\_OperatingSystem class you can modify the Description and ForegroundApplicationBoost properties.

How do you modify a property value on a CIM instance? The clue is in the question. You use Set-CimInstance. Let’s start by creating an environment variable:

PS> New-CimInstance -ClassName Win32\_Environment -Property @{

Name = 'PiAvar';

VariableValue = 'PiA 2017';

UserName = "$($env:COMPUTERNAME)\bpayette"}

Name UserName VariableValue

---- -------- -------------

PiAvar LAPTOPO2\bpayette PiA 2017

Modify the value assigned to the variable:

PS> Get-CimInstance -ClassName Win32\_Environment `

-Filter "Name = 'PiAvar'" |

Set-CimInstance -Property @{VariableValue='What about next year?'} `

-PassThru

Name UserName VariableValue

---- -------- -------------

PiAvar LAPTOPO2\bpayette What about next year?

The majority of the time when you’re accessing CIM data, you’re going to be working with remote machines. When using PowerShell remoting to access remote machines (see [chapter 11](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-11-93.xhtml#ch11)) you can create connections (remoting sessions) that can be used many times. An analogous situation exists when using the CIM cmdlets.

### 16.4.2 CIM Sessions

Creating and destroying links from your administration machine to a remote machine is an expensive proposition. You need to get the maximum return from creating that connection. PowerShell remoting enables you to create a persistent session you can reuse. In a similar manner, the CIM cmdlets have an option to use a CIM session.

#### Note

You can’t copy files over a CIM session as you can in a PowerShell v5 remoting session.

A CIM session is similar to a PowerShell remoting session in that it’s based on WS-MAN (by default) but is designed to be utilized by the CIM-based cmdlets and connects to a different endpoint (the WMI provider).

You create CIM sessions by passing one or more computer names to the New-CimSession cmdlet:

PS> $computers = 'W16DSC01', 'W16AS01'

PS> $cs = New-CimSession -ComputerName $computers

The CIM session object contains the computer name and the protocol:

PS> Get-CimSession -ComputerName W16DSC01

Id : 1

Name : CimSession1

InstanceId : 864bb2cf-3b08-4d65-8d0f-00f857f0a7a9

ComputerName : W16DSC01

Protocol : WSMAN

The session information is passed to the cmdlet through the -CimSession parameter:

PS> Get-CimInstance -CimSession $cs -ClassName Win32\_OperatingSystem |

select SystemDirectory, BuildNumber, Version, PSComputerName

SystemDirectory BuildNumber Version PSComputerName

--------------- ----------- ------- --------------

C:\Windows\system32 9600 6.3.9600 W16DSC01

C:\Windows\system32 9600 6.3.9600 W16AS01

CIM sessions need WS-MAN 3.0 (introduced with PowerShell v3). If you try to access a machine running PowerShell v2 (which used WS-MAN 2.0), you’ll get an error:

PS> $cs2 = New-CimSession -ComputerName W8R2STD01

PS> Get-CimInstance -CimSession $cs2 -ClassName Win32\_OperatingSystem |

select SystemDirectory, BuildNumber, Version, PSComputerName

Get-CimInstance : The WS-Management service cannot process the request. A

DMTF resource URI was used to access a non-DMTF class. Try again using a

non-DMTF resource URI.

At line:1 char:1

+ Get-CimInstance -CimSession $cs2 -ClassName Win32\_OperatingSystem | s ...

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : NotSpecified: (root\cimv2:Win32\_OperatingSystem:String) [Get-CimInstance], CimException

+ FullyQualifiedErrorId : HRESULT 0x80338139,Microsoft.Management.Infrastructure.CimCmdlets.GetCimInstanceCommand

+ PSComputerName : W8R2STD01

The problem is that PowerShell v2 used WS-MAN 2.0. You can overcome this issue by creating a DCOM-based CIM session:

PS> $opt = New-CimSessionOption -Protocol Dcom

PS> $csd = New-CimSession -ComputerName W8R2STD01 -SessionOption $opt

PS> $csd

Id : 4

Name : CimSession4

InstanceId : 2de6064f-f018-4b71-8d80-95fe2413089c

ComputerName : W8R2STD01

Protocol : DCOM

The DCOM-based CIM session is used in exactly the same manner as a WS-MAN session:

PS> Get-CimInstance -CimSession $csd -ClassName Win32\_OperatingSystem |

select SystemDirectory, BuildNumber, Version, PSComputerName

SystemDirectory BuildNumber Version PSComputerName

--------------- ----------- ------- --------------

C:\Windows\system32 7601 6.1.7601 W8R2STD01

Once you have a CIM session created, the CIM cmdlets will use WS-MAN- and DCOM-based sessions together:

PS> Get-CimInstance -CimSession ($cs + $csd) `

-ClassName Win32\_OperatingSystem |

select SystemDirectory, BuildNumber, Version, PSComputerName

SystemDirectory BuildNumber Version PSComputerName

--------------- ----------- ------- --------------

C:\Windows\system32 7601 6.1.7601 W8R2STD01

C:\Windows\system32 9600 6.3.9600 W16DSc01

C:\Windows\system32 9600 6.3.9600 W16AS01

#### Note

You can create a DCOM-based CIM session to machines running WS-MAN 3.0 if required.

Many of the cmdlets in Windows 8.0 and later are created using Cmdlet Definition XML (CDXML). The cmdlets in the NetAdapter and NetTCPIP modules are good examples.

#### Note

CDXML-based cmdlets are created using the cmdlets-over-objects technology introduced in PowerShell v3. In this case, you wrap the CIM class in the appropriate XML and publish as a PowerShell module. The CIM class must be on the remote machine for these cmdlets to work remotely.

If you look at the syntax of those cmdlets, you’ll see that they have a -CimSession parameter but don’t have a -ComputerName parameter. This is an artifact of the way they’re created. You have to use CIM sessions when using these cmdlets against remote machines. You can use a computer name as a value to the -CimSession parameter—it creates a session to the remote machine, executes the command, and then removes the session. If you’re making multiple CIM calls to the same machine, it’s more efficient to use a CIM session.

That concludes our investigation of using CIM through PowerShell and closes this chapter.

## 16.5 Summary

* PowerShell providers supply a file system–like experience for other data stores.
* Providers are exposed as PowerShell drives.
* Core cmdlets work across providers.
* The LiteralPath parameter suppresses pattern-matching behavior.
* Use the \*Content cmdlets to work with text files.
* Get-Content has -Head and -Tail parameters to read the beginning and end of files.
* Advanced string handling can be performed using the [string] and [regex] classes.
* Use Select-String to search string data.
* XML documents have to be created manually.
* Select-Xml uses XPath queries.
* ConvertFrom-String can convert text output from legacy applications to objects.
* COM objects can be accessed from PowerShell.
* COM processing speed is greatly increased in PowerShell v5.
* The CIM cmdlets should be used in preference to the older WMI cmdlets.
* Use Get-CimClass to discover CIM classes, methods, and properties.
* CIM sessions provide a persistent connection to remote machines.
* CIM sessions use WS-MAN by default; DCOM is available as an option for connecting to PowerShell v2 systems.

In the next chapter, we’ll continue our investigation of how PowerShell works with data when we look at using .NET and events.

## Chapter 17: Working with .NET and Events

### Overview

This chapter covers

* .NET and PowerShell
* Real-time events

*I love it when a plan comes together!  
Col. John “Hannibal” Smith, The A-Team*

The good news is that PowerShell is .NET -based and works with .NET objects. The not-quite-so-good news is that not all of .NET is immediately available when you open PowerShell. Some .NET functionality is available through cmdlets—for the rest you need to access the .NET classes in your code. PowerShell doesn’t load the entire .NET framework, so you’ll need to load assemblies before you can use them. Once an assembly is loaded, you have access to the rich .NET functionality, including creating graphical applications in PowerShell.

**Note**

PowerShell v6 uses .NET core which has further restrictions as described in the appendix.

Windows is an event-based system. You can use PowerShell to access events from a number of sources. Your scripts can then either display information about the event or take action based on the event.

We’ll start with .NET before moving on to events.

## 17.1 .NET and PowerShell

The original PowerShell concept was to have cmdlets for every task, but that goal wasn’t achievable in the time frame available for the release of PowerShell v1. Instead, the team made the decision to make it easier to work directly with the .NET Framework. That way, although it might not be as easy to do everything the way the team wanted, at least it would be possible.

In retrospect, this may have been one of the best things to happen to PowerShell. Not only did the team backfill their original scenarios, but the set of problem domains (such as creating GUIs) in which PowerShell was applicable greatly exceeded original expectations.

### 17.1.1 Using .NET from PowerShell

We’re assuming that you have a basic understanding of .NET. If you’re new to .NET or need a refresher, we recommend you read <http://mng.bz/RIvK> before reading the rest of this chapter. The basic arrangement of entities in .NET is as follows: members (properties, methods, and so on) are contained in types (classes, structs, and interfaces) which are, in turn, grouped into namespaces.

The arrangement of types into classes and namespaces is called logical type containment. You also need to understand physical type containment. Where do these collections of types live on a computer? This organization is done through the assemblies we mentioned earlier. An assembly is a file stored somewhere so that the program loader can find it when needed. Each assembly contains the definitions for one or more types. Because a set of types is contained in an assembly, clearly the set of assemblies that’s loaded determines the complete set of types available to you. PowerShell loads most of the assemblies you’ll need for day-to-day work by default when it starts, but sometimes (like when you want to do GUI programming) you’ll have to load additional assemblies.

#### Versioning and Assemblies

With .NET, Microsoft tried to solve some of the problems with assemblies, in particular the issue of versioning of DLLs. In effect, an assembly is a DLL with additional metadata in the form of an assembly manifest.

This assembly manifest lists the contents of the DLL as well as the name of the DLL. The full (or strong) name for an assembly is a complex beast and warrants some discussion. To try to solve some of the identity and versioning problems, .NET introduced the idea of a strong name. As well as the assembly filename, a strong name uses public key cryptography to add information that will allow you to validate the identity of the DLL author. When a .NET program is linked against a strong-named assembly, it will run only if exactly the same assembly it was linked against is present. Replacing the file won’t work, because the strong name will be wrong.

One more thing that’s included in the strong name is the version number. The result is that when the DLL is loaded, the correct version must always be loaded even if later versions are available. But it also means that to service the assembly to fix bugs, you can’t change the version number of the assembly because the version number is part of the strong name. You end up with two versions of an assembly with the same version number. The net effect of all of this is that .NET didn’t solve the versioning problem—it merely moved things around a bit.

#### The Default PowerShell Assemblies

Now let’s talk about how PowerShell finds types and assemblies. All compiled programs contain a list of assemblies needed for the program to execute. This list is created as part of the linking phase when the program is compiled. When the program executes, the referenced assemblies are loaded automatically as needed. When the system tries to locate a required assembly, the loader performs a process called probing to find that assembly. It looks in a number of places automatically; the most important one is the global assembly cache (GAC). If an assembly has been installed in the GAC, you don’t have to care where it is—the system will find it for you as long as you know its name.

Because the PowerShell interpreter is a compiled program, it also contains a list of required assemblies. Through the automatic loading mechanism, all these assemblies and the types they contain are available to PowerShell scripts by default.

You can view the assemblies PowerShell loads by default by opening a new PowerShell console (ensures only defaults are loaded) and running:

PS> [System.AppDomain]::CurrentDomain.GetAssemblies() |

sort Fullname | select Fullname

The AppDomain class is .NET’s way of encapsulating an isolated execution environment. It’s similar in some ways to PowerShell sessions but even more isolated. For example, each AppDomain can have its own set of assemblies, whereas PSSessions all share the same assemblies. The static CurrentDomain property lets you access the domain you’re executing in, and GetAssemblies() gives you the list of assemblies currently loaded into the AppDomain.

#### Note

You’ll see an extended set of assemblies if you run the code in PowerShell ISE as opposed to the console.

Once you have the list of assemblies, you can use the GetTypes() and GetExportedTypes() methods on each assembly object to get all the types in that assembly. The GetExportedTypes() method gives you all the public types, which is usually what you want. GetTypes() returns both public and private types, which is primarily useful for exploring how things are organized below the public façade. The function in the following listing gets the full names of all of the public types in each assembly and matches them against the pattern provided in the function argument (which defaults to matching everything).

#### Listing 17.1: Getting exported types from .NET assemblies

function Get-Type {

[CmdletBinding()]

param (

[string]$Pattern='.'

)

[System.AppDomain]::CurrentDomain.GetAssemblies() |

Sort-Object FullName |

foreach{

$asm = $psitem

Write-Verbose $asm.Fullname

switch ($asm.Fullname) {

{$\_ -like

https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/images/enter8.jpg 'Anonymously Hosted DynamicMethods Assembly\*'}{break}

{$\_ -like

https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/images/enter9.jpg 'Microsoft.PowerShell.Cmdletization.GeneratedTypes\*'}

https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/images/enter10.jpg {break}

{$\_ -like 'Microsoft.Management.Infrastructure.

https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/images/enter11.jpg UserFilteredExceptionHandling\*'}

https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/images/enter12.jpg {break}

{$\_ -like 'Microsoft.GeneratedCode\*'}{break}

{$\_ -like 'MetadataViewProxies\*'}{break}

default {

$asm.GetExportedTypes() |

Where-Object {$\_ -match $Pattern} |

Select-Object @{N='Assembly';

E={($\_.Assembly -split ',')[0]}},

IsPublic, IsSerial,FullName, BaseType

}

}

}

}

Use the function like this:

PS> Get-Type -Pattern '^system\.timers' |

Format-Table Assembly, IsPublic, Fullname

Assembly IsPublic FullName

-------- -------- --------

System True System.Timers.ElapsedEventArgs

System True System.Timers.ElapsedEventHandler

System True System.Timers.Timer

System True System.Timers.TimersDescriptionAttribute

Add the -Verbose switch on Get-Type to see the list of assemblies that are scanned as well as the results.

#### Dynamic Assembly Loading

Automatic loading applies only to compiled programs like Notepad.exe or PowerShell.exe because it depends on the required assembly list contained in the executable. PowerShell scripts are interpreted and have no compile or “static” link phase, so if you want to make sure that an assembly you need is loaded, you have to explicitly load it. In [chapter 10](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-10-82.xhtml#ch10), you saw how to do this with module manifests; you add the list of required assemblies to the RequiredAssemblies manifest element. In effect, module manifests are the dynamic equivalent to the static manifest found in an assembly. But with simple scripts you don’t have a manifest, so in this case you’ll use the Add-Type cmdlet—the “Swiss Army knife cmdlet”—for dealing with assemblies and compiled code.

#### Note

You’ll see numerous examples where [system.reflection.assembly] ::LoadWithPartialName is used to load an assembly. This is a hangover from PowerShell v1 and shouldn’t be used because the LoadWithPartialName method is obsolete.

You can dynamically load assemblies by name. You can even use wildcards in the assembly name (but an error is generated if more than one assembly filename matches the pattern). For example, to load the Windows Forms assembly (winforms) that’s in System.Windows.Forms, instead of the full name, you can use:

PS> Add-Type -AssemblyName System\*forms

This works because Add-Type has a fixed list of short names that correspond to specific versions of the .NET Framework assemblies. Add-Type will allow you to use the short name only for assemblies that are on this list. If it’s not on the list, you have to use the strong name for the assembly. For winforms, the strong name looks like

"System.Windows.Forms, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

which is a bit unwieldy. Still, as long as you stick to the assemblies Microsoft ships with Windows, you can use the short names and a wildcard. If you choose a non-Windows assembly, you have to use the full name.

#### Creating your own types

You can create your own types using C#, Visual Basic, or JScript and then use Add -Type to compile them into your PowerShell session.

In PowerShell v5 you can create classes in PowerShell, which we cover in [chapter 19](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-19-143.xhtml#ch19).

#### Creating Instances of Types

Now that you can find types, you generally need to create instances of these types to use their properties and methods (although there are some types such as [System.Math] that have only static members and so don’t require instantiation). For example, before you can search using the [regex] type, you need to create an instance of that type from a pattern string. As you saw in earlier chapters, you can use the New-Object cmdlet to create instances of types in PowerShell.

This cmdlet takes the name of the type to create, a list of parameters to pass to the type’s constructor, and a hashtable of property name/values to set on the object once it has been constructed.

#### The New-Object -Property Parameter

The -Property parameter allows individual properties to be set on the object after it has been constructed. In many cases, doing so can greatly simplify the code needed to completely initialize an object. For a simple example, let’s create a Timer object:

PS> New-Object -TypeName System.Timers.Timer -Property @{

AutoReset = $true

Interval = 500

Enabled = $true

}

AutoReset : True

Enabled : True

Interval : 500

Site :

SynchronizingObject :

Container :

In this example, you’re creating the object and then setting three properties—AutoReset, Interval, and Enabled—in a single statement. Without -Property, you’d have to create an intermediate variable and use four statements. We need to explain some more cautions.

#### A Word of Caution about Using New-Object

Although the signature for the New-Object cmdlet is pretty simple, it can be more difficult to use than you might think. People who are accustomed to programming in languages such as C# have a tendency to use this cmdlet like the new operator in those languages. As a consequence, they tend to write expressions like this:

PS> $x = 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'

PS> New-Object string($x,1,3)

Unfortunately, writing the expression this way obscures the fact that it’s a cmdlet, making things confusing. It’ll work fine, but it looks too much like a function call in other programming languages, and that leads people to misinterpret what’s happening. The syntax for New-Object is as follows:

New-Object [-TypeName] <String> [[-ArgumentList] <Object[]>]

That means the previous example could be written like this:

PS> New-Object -TypeName string -ArgumentList $x,1,3

The comma notation indicates an argument that’s passed as an array. This is equivalent to

PS> $constructor\_arguments= $x,1,3

PS> New-Object string $constructor\_arguments

#### Note

You’re not wrapping $constructor\_arguments in yet another array. If you want to pass an array as a single value, you need to do it yourself and write it in parentheses with the unary comma operator.

#### Working with Generic Types

With version 2.0 of .NET, a feature was added to the CLR type system called generic types (or generics). Generics introduce the idea of a type parameter. Instead of passing objects as arguments when creating an instance of type, generics also require you to pass in type parameters that are used to determine the final types of some part of the object. This concept is rather confusing if you haven’t encountered it. As usual, an example should make things clearer.

Generics are easiest to understand when you talk about creating collections. Before the introduction of generics, if you wanted to create a collection class, you had to either write a new version of the class for each type of object you wanted it to store or you had to allow it to hold any type of object, which meant you had to do your own error checking. With generics, you can write a collection that can be constrained to contain only integers or strings or hashtables. Let’s look at examples.

We’ll start by creating a list—specifically, a list of integers. To do this, you need to know the base type of the collection and the type parameter you need to pass when creating an instance of the collection. The base type you’re going to use is System.Collections.Generic.List, which takes a single type argument. To create an instance of the collection, you pass the closed type name to New-Object. By closed, we mean that a concrete type has been specified as the type parameter. For a collection of integers, this looks like

PS> $ilist = New-Object System.Collections.Generic.List[int]

where the name in the square brackets is the type parameter. You can use other types as well. To create a list of strings, you’d write

PS> $slist = New-Object System.Collections.Generic.List[string]

You can even use generics in the type parameter:

PS> $nlist = New-Object `

System.Collections.Generic.List[System.Collections.Generic.List[int]]

This example defines a list of lists of integers. In general, nested generic types are discouraged because they quickly become difficult to understand.

So far, we’ve dealt with only a single type parameter, but generics can take as many type parameters as are needed. For example, a generic dictionary, which is similar to our old friend the hashtable, takes two type parameters: the type of the key and the type of the value. This looks like

PS> $stoi = New-Object 'System.Collections.Generic.Dictionary[string,int]'

Notice that this time you have to put quotes around the type name—otherwise, the comma between the two type parameters would cause the type name to be treated as separate parameters.

With all this time we’ve spent playing the .NET trivia challenge game, we’re sure heads are buzzing and coffee is being desperately sought. In the remainder of this section, we’ll look at how you can apply some of the things you’ve learned to build more interesting applications.

### 17.1.2 PowerShell and GUIs

The full name of the PowerShell package is Windows PowerShell. In this section, we’ll look at the Windows part of the name. You can do GUI programming with PowerShell, as you’ll see.

We’re going to look at both WinForms and Windows Presentation Foundation (WPF) because the framework used in a particular scenario will depend on a number of criteria. First and foremost, WPF can’t be used with PowerShell v1 because v1 doesn’t support the single-threaded apartment (STA) threading model which allows a thread waiting on a time-consuming operation to allow another thread to run. If you need to write a UI for use in an STA environment, WinForms is your only choice.

#### Note

PowerShell v5 starts the shell using STA by default. This change was introduced in PowerShell v3.

Second, the tools you have available will influence your choice. There are now GUI designers that support using WinForms with PowerShell. This may make WinForms the better, easier, and faster way to do things. WPF, conversely, makes it much easier to create rich, modern UIs. It also supports clean separation of business logic and presentation, allowing the look of the application to be changed without requiring changes to the underlying scripts. You’ll see more of these details as we look at each framework. Finally, WinForms has been part of .NET since the beginning, whereas WPF was added with .NET 3.0. If you need your GUI to run on a .NET 2.0–only system, you should look at using WinForms.

Each of these libraries provides a framework and collection of utility classes for building graphical application UIs. Let’s see what you can do with these libraries. We’ll begin by looking at WinForms.

#### PowerShell and WinForms

The core concepts in WinForms are controls, containers, properties, and events. A control is an element in a UI—buttons, list boxes, and so on. Most controls, like buttons, are visible controls that you interact with directly, but there are some controls, such as timers, that aren’t visible yet still play a role in the overall user experience. Controls have to be laid out and organized to present a GUI. This is where containers come in. Containers include things such as top-level forms, panels, splitter panels, tabbed panels, and so on. Within a container, you can also specify a layout manager which determines how the controls are laid out within the panel. Properties are regular properties, except that they’re used to set the visual appearance of a control. You use them to set things such as the foreground and background colors or the font of a control.

The final piece in the WinForms architecture is the event. Events are used to define the behavior of a control both for specific actions, such as when a user clicks the Do It button, as well as when the container is moved or resized and the control has to take some action. Like everything else in .NET (and PowerShell), events are represented as objects. For WinForms, the most common type of event is System.EventHandler. For PowerShell, anywhere an instance of System.EventHandler is required you can use a scriptblock. If you want a particular action to occur when a button is clicked, attach a scriptblock to the button click event.

#### EventHandler arguments

For an event handler to do its job, it requires information about the event that caused it to be invoked. You saw a similar pattern with ForEach-Object and Where-Object, where the value the scriptblock operated on was passed using the automatic variable $\_. The EventHandler integration in PowerShell follows the same basic pattern. In .NET, when an EventHandler is invoked, it’s passed two arguments: the object that fired the event and any arguments that are specific to that event. The signature of the method that’s used to invoke an event handler looks like this:

void Invoke(System.Object, System.EventArgs)

These values are made available to the scriptblock handling the event using the automatic variables $this and $\_. The variable $this contains a reference to the object that generated the event, and $\_ holds any event-specific arguments that might have been passed. In practice, you don’t need these variables most of the time because of the way variables in PowerShell work. With global, script, and module scopes, you can usually access the objects directly. Still, it’s good to be aware of them in case you need them.

Many elements in building a Windows Forms application are repeated over and over. If you’re working in an environment such as Visual Studio, the environment takes care of generating the boilerplate code. But if you’re building a form using Notepad, you need to be a bit more clever to avoid unnecessary work. Let’s build a module containing a number of convenience functions that make it easier to work with WinForms. We’ll call this module WPIAForms. If this module is placed somewhere in your module path, then you can use it by including the line

Import-Module WPIAForms

at the beginning of your script.

#### Listing 17.2: The WPIAForms.psm1 module

Add-Type -Assembly System.Drawing, System.Windows.Forms 1

function New-Size 2

{

param (

[Parameter(mandatory=$true)] $x,

[Parameter(mandatory=$true)] $y

)

New-Object System.Drawing.Size $x,$y

}

function New-Control 3

{

param (

[Parameter(mandatory=$true)]

[string]

$ControlName,

[hashtable] $Properties = @{}

)

$private:events = @{}

$private:controls = $null

foreach ($pn in "Events", "Controls") 4

{

if ($v = $Properties.$pn)

{

Set-Variable private:$pn $v

$Properties.Remove($pn)

}

}

$private:control = if ($Properties.Count) { 5

New-Object "System.Windows.Forms.$ControlName" `

-Property $Properties }

else {

New-Object "System.Windows.Forms.$ControlName" }

if ($controls) { 6

[void] $control.Controls.AddRange(@(& $controls)) }

foreach ($private:en in $events.keys) 7

{

$method = "add\_$en"

$control.$method.Invoke($events[$en])

}

if ($control -eq "form") { 8

$c.add\_Shown({ $this.Activate() }) }

$control 9

}

* 1 Load required assemblies
* 2 Create Size objects
* 3 Create controls
* 4 Extract events and controls from the hashtable
* 5 Construct a control object
* 6 Add child controls
* 7 Bind event handlers
* 8 Ensure the form is visible
* 9 Return the configured control

The first thing a WinForms module should do is make sure that the necessary assemblies are loaded 1. (Remember that trying to load an assembly multiple times is harmless.)

Next, you define a convenience function 2 for creating Size objects. Like many helper functions, it hides the long type names used to construct the objects.

Then you come to the heart of the module: the New-Control function 3. This function is used to construct all the controls for your UI. It takes as arguments the name of the WinForms control class to instantiate and a hashtable containing three types of entries:

* Simple properties to set on the control
* An Events hashtable specifying which control events you want to handle
* A scriptblock used to create the child controls for this form

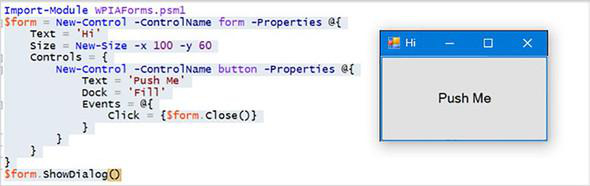
The function iterates over the keys in the hashtable 4, looking to extract the Controls and Events members because they aren’t simple properties on the object you’re creating. The scriptblock in the Controls member will be evaluated, and any control objects it returns will be added as children of the current control. The Events member requires more complex processing. It’s also a hashtable, but in this case the keys are the names of control events, and the values are the scriptblocks to bind to those events.

Once the two special members have been extracted, the function passes the cleaned-up hashtable to the -Property parameter on New-Object 5 to initialize the control. Unfortunately, there’s an annoying limitation on -Property: If the value passed to New-Object is either $null or empty, it will error out. This necessitates wrapping the call to New-Object in an if statement so that -Property gets used only when the hashtable is not empty.

Now that the control object exists, add any child controls that were extracted 6 and bind any event handlers that were specified 7. One additional event handler is added to ensure that the window is visible 8. Finally, the completely configured control object is returned 9.

Although there doesn’t seem to be much to this library, it can significantly clarify the structure of the application you’re building. Try it out by re-implementing the one-button example and see what it looks like. The result is shown in [figure 17.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-171-132.xhtml#ch17fig01).

Figure 17.1: An example using the WPIAForms module. Both the code and the resulting window are shown here



The resulting code isn’t any shorter, but the hierarchical structure of the form is much more obvious. The top-level form is created using New-Control and sets the title to “Hi” and the size of the form to 100 x 60. The Controls member scriptblock creates the child controls for the form. In this case you’re adding a Button object, and again you use New-Control to create the object, set the Text and Dock properties, and define the Click event handler. Notice that at no point did you have to write any conditional loops—instead of describing how to build the form, you’ve declared what you want. In effect, you’ve created a simple DSL for defining WinForms-based UIs.

#### Note

A number of GUI builders on the market support building WinForms UIs in PowerShell, including SAPIEN Technologies PowerShell Studio and iTripoli’s Admin Script Editor (which has an integrated PowerShell forms designer and is now unsupported freeware). Both of these tools provide sophisticated PowerShell authoring environments as well as (or with) the forms designer. GUI builders eliminate most of the manual layout and UI construction code.

Let’s see where you’ve ended up. In the previous example, you invented a rather limited DSL for building GUIs in a declarative way. Clearly the ability to separate UI structure from the implementation logic is compelling, so it would be nice if, rather than inventing your own language, you could use an existing GUI definition language. In practice, this is exactly what the WPF is. Therefore, we’re going to spend time seeing how WPF can simplify building UIs in PowerShell.

#### PowerShell and the WPF

In this section, you’ll learn how to use WPF from PowerShell to construct GUIs. WPF takes a different approach to constructing a GUI compared to WinForms. With WPF the UI is written declaratively using an XML-based markup language called XAML (Extensible Application Markup Language). The approach used in WPF is similar to the DSL you wrote as well as to the way HTML works: you describe the basic components, and the framework handles all the construction details. An important aspect of the design of WPF is that the UI description is decoupled from the UI logic. This separation of appearance and behavior aligns with well-established best practices for UI design (such as coders write code and design specialists do design).

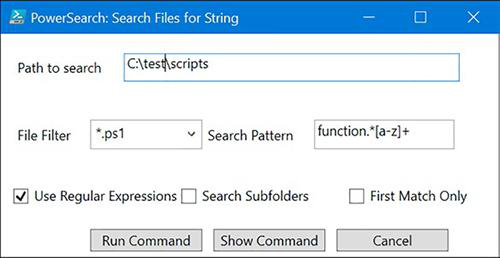
You’ll see how this all works by building a simple GUI front end to some PowerShell commands. We’ll cover only a fraction of the features of WPF—just enough to accomplish our goal of quickly building a simple UI. First, you’ll have to satisfy a few prerequisites before you can use WPF from PowerShell.

Although WPF has been around as long as PowerShell, in PowerShell v1 you weren’t able to use WPF without a lot of tricks. That’s because WPF can only be called from an STA-mode thread (yes, here it is again). With PowerShell v2 and later, this limitation ceased to be an impediment. (And in the ISE, which is a WPF application, you always run in STA mode, so by default everything will work.)

The other thing you need to do to use WPF in your scripts is to load the WPF assemblies, PresentationCore and PresentationFramework, using Add-Type. With these prerequisites out of the way, you can start working on our example project.

The goal of this exercise is to create a GUI front end to the Get-ChildItem and Select-String cmdlets using WPF. You want novice users to be able to execute a file search without having to be experts in PowerShell. A screen shot of the desired UI is shown in [figure 17.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-171-132.xhtml#ch17fig02).

Figure 17.2: A dialog box that front-ends the PowerShell Get-ChildItem and Select-String cmdlets, allowing users to search with PowerShell even if they don’t know the language



In this form, the user can specify the path to search (defaulting to the current directory), the extension of the files to search, and the pattern to use when searching the file text. By default, regular expressions will be used in the text search, but an option is provided to suppress this. There are also options to indicate that subfolders should be searched as well and that only the first match in each file may be returned. At the bottom of the dialog box are buttons to run or cancel the search. There’s also a button that will display the command to be run before executing it—a useful mechanism for learning PowerShell.

Although this is a simple dialog box, it would be annoying to implement with WinForms because of the manual control layout required. WPF uses XAML to describe the interface. The XAML text for the interface you’re going to create is shown next.

#### Listing 17.3: The search.xaml file declaring the file search interface

<Window 1

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" 1

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" 1

Title="PowerSearch: Search Files for String" 1

SizeToContent="WidthAndHeight" > 1

<DockPanel>

<StackPanel HorizontalAlignment="Left" Orientation="Horizontal" 2

Width="425" DockPanel.Dock="Top" Margin="10,17,10,17">

<Label Width="100" >Path to search</Label> 3

<TextBox Name="Path" Width="300" >Add Row</TextBox> 4

</StackPanel>

<StackPanel HorizontalAlignment="Left" Orientation="Horizontal" 5

Width="425" DockPanel.Dock="Top" Margin="10,17,10,17">

<Label Width="70" >File Filter</Label>

<ComboBox Name="FileFilter" Width="100" IsEditable="True"> 6

\*.ps1

</ComboBox>

<Label Width="100" >Search Pattern</Label>

<TextBox Name="TextPattern" Width="125" >

function.\*[a-z]+

</TextBox>

</StackPanel>

<StackPanel HorizontalAlignment="Left" Orientation="Horizontal"

Width="425" DockPanel.Dock="Top" Margin="10,17,10,17">

<CheckBox Name="UseRegex" Width="150" >

Use Regular Expressions

</CheckBox>

<CheckBox Name="Recurse" Width="150" >

Search Subfolders

</CheckBox>

<CheckBox Name="FirstOnly" Width="150" >

First Match Only

</CheckBox>

</StackPanel>

<StackPanel HorizontalAlignment="Left" Orientation="Horizontal"

DockPanel.Dock="Top" Margin="75,5,5,5">

<Button Width="100" Name="Run" Margin="5,0,5,0" > 7

Run Command

</Button>

<Button Width="100" Name="Show" Margin="5,0,5,0" >

Show Command

</Button>

<Button Width="100" Name="Cancel" Margin="5,0,5,0" >

Cancel

</Button>

</StackPanel>

</DockPanel>

</Window>

* 1 Create the top-level window.
* 2 Create a StackPanel to hold controls
* 3 Add a Label to the StackPanel.
* 4 Add a named TextBox to the StackPanel.
* 5 Create another StackPanel for the next row.
* 6 Add a ComboBox for the file filter.
* 7 Add dialog buttons to the bottom.

Looking through the XAML code, you see many things that are familiar from the WinForms examples: Label controls, TextBoxes, Buttons, and so on. This means you don’t have to learn a lot of new concepts, rather a new way to describe how they should be put together. In this UI description, the dialog box is constructed as a set of rows of controls. A StackPanel layout control is used to arrange the elements in each row, and a DockPanel holds all the rows.

Let’s look at one of the control declarations in detail. The XAML that declares the Run button looks like this:

<Button Width="100" Name="Run" Margin="5,0,5,0" >

Run Command

</Button>

By inspecting the text, you can see that you’re creating a Button control, setting the Width property on that control to 100, and setting the control Margin property with values for left, top, right, and bottom margins. Of particular importance is the Name property, which lets you associate a unique name string with the control. You’ll need this information later when you’re binding actions to the controls.

This XAML document describes what your form will look like but doesn’t say anything about how it behaves.

#### Note

At this point, the XAML experts in the audience will be shouting that, in fact, many elements in XAML do let you describe behaviors (animations, triggers, and such). These features are beyond the scope of this exercise, but you’re encouraged explore all the things that can be done with XAML. It’s amazing how much you can accomplish using markup.

You now have to attach your business logic to this markup. To display your form, you must load the XAML into the session and use it to create an instance of System.Windows.Window. The WPF framework includes utility classes to do most of the heavy lifting for this task. Once you have the UI object, you have to attach PowerShell actions to the controls. The following listing shows the script that does both of these things for you.

#### Listing 17.4: search.ps1: defining the file search behavior

Add-Type -Assembly PresentationCore,PresentationFrameWork 1

trap { break }

$mode = [System.Threading.Thread]::CurrentThread.ApartmentState

if ($mode -ne "STA")

{

$m = "This script can only be run when powershell is " +

"started with the -sta switch."

throw $m

}

function Add-PSScriptRoot ($file) 2

{

$caller = Get-Variable -Value -Scope 1 MyInvocation

$caller.MyCommand.Definition |

Split-Path -Parent |

Join-Path -Resolve -ChildPath $file

}

$xamlPath = Add-PSScriptRoot search.xaml

$stream = [System.IO.StreamReader] $xamlpath 3

$form = [System.Windows.Markup.XamlReader]::Load(

$stream.BaseStream)

$stream.Close()

$Path = $form.FindName("Path") 4

$Path.Text = $PWD

$FileFilter = $form.FindName("FileFilter") 5

$FileFilter.Text = "\*.ps1"

$TextPattern = $form.FindName("TextPattern")

$Recurse = $form.FindName("Recurse")

$UseRegex = $form.FindName("UseRegex") 6

$UseRegex.IsChecked = $true

$FirstOnly = $form.FindName("FirstOnly")

$Run = $form.FindName("Run") 7

$Run.add\_Click({

$form.DialogResult = $true

$form.Close()

})

$Show = $form.FindName("Show")

$Show.add\_Click({Write-Host (Get-CommandString)})

$Cancel = $form.FindName("Cancel")

$Cancel.add\_Click({$form.Close()})

function Get-CommandString 8

{

function fixBool ($val) { '$' + $val } 9

"Get-ChildItem $($Path.Text) ``

-Recurse: $(fixBool $Recurse.IsChecked) ``

-Filter '$($FileFilter.Text)' |

Select-String -SimpleMatch: (! $(fixBool $UseRegex.IsChecked)) ``

-Pattern '$($TextPattern.Text)' ``

-List: $(fixBool $FirstOnly.IsChecked)"

}

if ($form.ShowDialog()) 10

{

$cmd = Get-CommandString

Invoke-Expression $cmd

}

* 1 Load the WPF assemblies.
* 2 Compute the path to the XAML file.
* 3 Load the XAML that constructs the UI.
* 4 Find and set the Path control to $PWD.
* 5 Set the default file filter extension.
* 6 Set up the CheckBox controls.
* 7 Bind the button Click actions.
* 8 Build the command string
* 9 Format Booleans so “True” becomes $true.
* 10 Show the form and wait.

As was the case with the contents of the XAML file, many elements in this script should be familiar from the WinForms examples. To add an action to a button, you use the add\_Click() event method, as you did with WinForms. You use the Text property on TextBox controls to get and set the contents of those controls. CheckBoxes have an IsChecked property, as was the case with WinForms. The biggest difference here is that, instead of binding actions as you construct the form, the XAML loader does all the construction and returns the completed object. You then have to find the controls by name to be able to access them. In practice, this turns out to be pretty simple. Once you’ve located the control objects, everything else is much the same as it was with WinForms. The Get-CommandString function is used to generate a string containing the PowerShell command that will perform the search. This function uses the retrieved control objects along with string expansion to produce a complete command.

#### Advantages of Using WPF

The biggest advantage of using WPF is the separation of UI description from UI behavior. By not mixing code and markup, each piece becomes simpler and can be modified fairly independently. For example, because you’re identifying the controls by name, it doesn’t matter where they get moved around in the form—you’ll still get the correct control when you ask for it by name.

The other big advantage to this separation of concerns is that you can now use all the WPF XAML GUI builders with PowerShell. Unlike WinForms, where the tools needed to know PowerShell to work, XAML is XAML, so the programming language (for the most part) doesn’t matter, and the UI can be designed independently, decoupled from any code. This also means that the UI can be designed by an expert UI designer and the code added by an expert scripter. Finally, the higher-level nature of the WPF framework means that more effective PowerShell GUI frameworks can be created.

#### PowerShell Frameworks for WPF

Inspired by possibilities that arise from the combination of PowerShell and WPF/XAML, the PowerShell community has created a high-level library for building WPF GUIs in PowerShell. The library is called ShowUI and is a free download from <https://github.com/showui/showui>. As of this writing, the latest version is 1.5.

ShowUI is the result of merging the WPK library written by James Brundage (who was a member of the PowerShell team) and the PowerBoots library, originally written and coordinated by Joel “Jaykul” Bennett (who is a PowerShell MVP). ShowUI is packaged as a PowerShell module and provides a multitude of useful features.

And with that, we’ve finished our tour of .NET and what you can do with it. It’s time to turn our attention to working with events.

## 17.2 Real-Time Events

An event is exactly what it sounds like: something happens; for example, a file is changed, a button is clicked, or a process is started. Events on a Windows machine can be synchronous or asynchronous. In this section, we’ll spend more time working with synchronous events, but the majority of the material will focus on asynchronous events. Asynchronous event handling allows scripts to respond to real-world events in a timely manner. We’ll also explore the basic concepts of event-driven scripting, the PowerShell eventing model and infrastructure, and how to apply this feature.

### 17.2.1 Foundations of Event Handling

PowerShell supports three major categories, or sources, of events: .NET object events, CIM events, and engine events (events generated by PowerShell itself). But before we go into specific discussions on any of these topics, you need a common understanding of the concepts and terminology used in event-based programming.

In the .NET Framework (and therefore in PowerShell), events are tangible objects represented using classes. When you look at any class in the .NET Framework, you’ll see that, along with methods and properties, each class also exposes some events. These event members are the focus of our discussion of .NET eventing.

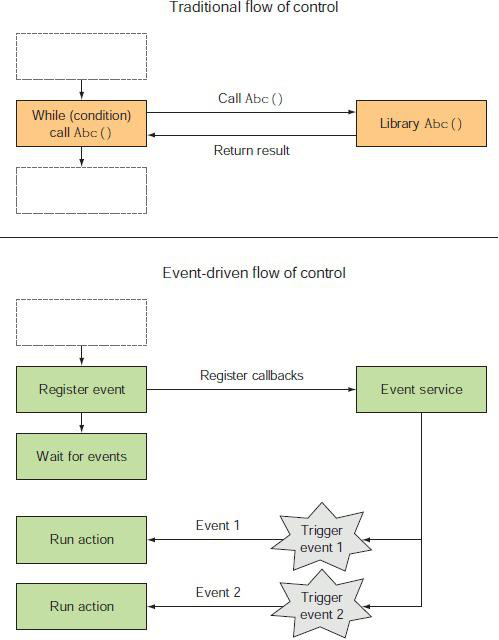
Now let’s talk about what an event is, and what makes event-based scripting different from traditional procedural scripting. The key difference with event-based scripting is that instead of an activity being executed as a result of an action in the script, a script (or at least a portion of it) is executed as a result of an action by the system. This pattern is sometimes called inversion of control, but it can be expressed more colorfully as don’t call me, I’ll call you.

#### Note

This way of characterizing event-based programming captures the essence of the model perfectly. Crispin Cowan (Linux Security and now Windows Security Guru Extraordinaire) suggested the “don’t call me, I’ll call you” definition as he and Bruce were hiking through the Cougar Mountains in Washington. Clearly, inspiration can arrive anywhere.

The traditional and event-driven flow control patterns are shown in [figure 17.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17fig03).

Figure 17.3: The normal flow of control in a script is compared to the flow in an event-based script



Look at the traditional flow of control illustrated in [figure 17.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17fig03). In the traditional model, the flow of control always belongs to the mainline of the program. If an action is required, the mainline program directly invokes that action. In contrast, with the eventing pattern, rather than directly initiating actions, the mainline program registers the set of actions with an event source and then goes to sleep. It never initiates any actions on its own. Instead, the event source is responsible for initiating actions as required. In this scenario you are, in effect, turning control over to the event service.

#### Note

In practice, we’ve been using this callback pattern all along, not only in GUIs. This is how the ForEach-Object and Where-Object cmdlets work: You pass action scriptblocks to the cmdlets, and the cmdlets take care of calling your code when it’s needed.

In other situations, the event service may be an active entity like another thread or process. In practice, real programs rarely restrict themselves to a single model but instead use different models at different times as appropriate.

### 17.2.2 Synchronous Events

The defining characteristic of synchronous eventing is that there’s never more than one action occurring at any given time. All the event-driven actions are synchronized, and no action is ever interrupted. This is the event-handling pattern used in most GUI frameworks like Windows Forms or Windows Presentation Foundation.

#### Synchronous Eventing in GUIs

In synchronous GUI frameworks, you create a collection of GUI elements and then register actions with these elements so that when the user does something like click a button, your actions will be executed. Once you’ve finished creating the GUI and registering the event actions, you hand control over to the framework, which will call the actions you defined when it needs to.

In PowerShell, for defining synchronous event handlers, you can usually attach a scriptblock directly to the event member on the object. In fact, you’ve already applied this pattern many times, as in the following, familiar example from [chapter 1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-1-8.xhtml#ch01):

Add-Type -AssemblyName System.Windows.Forms

$form = New-Object -TypeName System.Windows.Forms.Form

$button = New-Object -TypeName Windows.Forms.Button

$button.text='Push Me!'

$button.Dock='fill'

$button.add\_Click({$form.close()})

$form.Controls.Add($button)

$form.ShowDialog()

By now, you know that this code creates a button that will close its containing form when clicked. The line $button.add\_Click({$form.close()}) is where the event handler is attached, or bound, to the control. The Button object has a Click event, which fires when the button is clicked. To add the Click event handler, you call the add\_Click() method, passing in the scriptblock to execute. Because the add\_Click() method requires an argument of type System.EventHandler, PowerShell automatically wraps the scriptblock with a generated subclass of System.EventHandler. The System.EventHandler class is an example of what is called a delegate in .NET terminology.

#### Delegates and Delegation

In the GUI examples you saw in [section 17.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-171-132.xhtml#ch17lev1sec1), you set up control actions in the GUI by attaching event handlers to the controls in the UI. When you set up event handlers like this, you are, in effect, delegating the execution of that code to the UI and depending on it to call the code at the right time. Because this involves delegated actions, a logical name for these event handlers would be delegates—which is what they’re called in .NET.

Events are represented as members on a class. The delegate values you assign to event members are represented by types that derive from a common base type. In this case, the common base type is System.Delegate. Depending on the argument type for the target event member, the required event handler argument will be a specific subclass of Delegate. In the PowerShell world, the role of the delegate is always played by scriptblocks. But because scriptblocks don’t derive from System.Delegate, the PowerShell runtime has to synthesize Delegate wrapper classes for the scriptblock that match the argument type required by the event member. Let’s see how this works in a non-GUI example.

#### Note

PowerShell v1 supported only the single subclass of delegate, System.EventHandler, because of time restrictions. This type was chosen because it’s widely used in the .NET framework, in particular by the GUI frameworks. This meant that there were many useful things you could do even though you had only the one type. In v2 and later, the delegate-wrapping support is generalized to cover all types of delegates, so you no longer have to deal with the limitations of only one delegate type.

#### A Non-GUI Synchronous Event Example

Although the use of System.EventHandler is common in .NET, additional synchronous delegate types in the .NET Framework don’t follow the System.EventHandler pattern. PowerShell v2 and later versions have greatly improved support for delegate types, and the PowerShell runtime can automatically generate wrappers for any type of delegate.

#### Note

You can generate a wrapper for any event type, but you can’t always automatically infer what type to generate in all scenarios. For those cases, the use of an explicit cast is required to disambiguate things. When you cast a scriptblock to the target type, the correct wrapper can be synthesized.

In this example, we’ll look at how PowerShell’s enhanced delegate handling works. You’re going to use a scriptblock as the MatchEvaluator in a call to the static Replace() method on the [regex] class. View the overloads for the Replace() method by using this:

PS> [regex]::Replace

The overload of Replace() you’re interested in uses a delegate to do custom transformations during the replace operation. The signature for this method is

static string Replace(

string input,

string pattern,

System.Text.RegularExpressions.MatchEvaluator evaluator)

The first two arguments are the string to act on and the pattern to search for. The final argument is a delegate of type

[System.Text.RegularExpressions.MatchEvaluator]

Now, examine this type:

PS> [System.Text.RegularExpressions.MatchEvaluator] |

Format-List Name,FullName,BaseType

Name : MatchEvaluator

FullName : System.Text.RegularExpressions.MatchEvaluator

BaseType : System.MulticastDelegate

You can see that it derives from System.MulticastDelegate. Because delegates are invoked using the Invoke() method, by looking at this method’s signature you can see what parameters your scriptblock requires. Let’s see what this method looks like for the MatchEvaluator delegate (note the leading space in the ' Invoke' pattern, which reduces the set of matched members):

PS> [System.Text.RegularExpressions.MatchEvaluator] |

foreach {[string] ($\_.GetMembers() -match ' Invoke')}

System.String Invoke(System.Text.RegularExpressions.Match)

You see that the delegate takes a single parameter representing the matched text, so the scriptblock will look like this:

{param($match) ... }

Note that in this scriptblock definition, we omitted the type attribute for simplicity, and in practice it isn’t needed. The delegate signature guarantees that the scriptblock will never be called with the wrong argument types.

And now that you have the signature figured out, let’s find out what this method does. Looking up the MatchEvaluator class on MSDN, you see the following:

*You can use a MatchEvaluator delegate method to perform a custom verification or manipulation operation for each match found by a replacement method such as Regex.Replace(String, Match-Evaluator). For each matched string, the Replace method calls the MatchEvaluator delegate method with a Match object that represents the match. The delegate method performs whatever processing you prefer and returns a string that the Replace method substitutes for the matched string.*

For your purposes, this means that whatever the scriptblock returns will replace the matched substring. Let’s try it out. Write an expression that will replace all the characters in a string with their corresponding hex representation:

PS> $inputString = 'abcd'

PS> [regex]::replace($inputString, '.',

[System.Text.RegularExpressions.MatchEvaluator] {

param($match)

'{0:x4}' -f [int] [char]$match.value

}

)

0061006200630064

Inside the scriptblock, you take each argument character and then use the format operator to turn it into a set of four hexadecimal digits.

By now, you should be comfortable with synchronous events. Asynchronous events introduce a number of considerations that make handling them more complicated. But because asynchronous events are a much more realistic way to model the world, the ability to handle them in PowerShell is important in scenarios such as responding to alerts. Beginning with the next section, you’ll spend quite a bit of time mastering these event patterns and learning how to apply them to solve real problems.

### 17.2.3 Asynchronous Events

Asynchronous events are much trickier to deal with than their synchronous cousins. A synchronous event effectively runs on the same thread of execution as everything else. At no point are there ever two actions occurring at the same time. Everything happens deterministically, eliminating any collisions or consistency/coherency issues.

Unfortunately, that model doesn’t match the way much of the real world works. Real-world events don’t occur in a strict deterministic order—they happen when they happen, interrupting whatever else might be going on at that time. This type of concurrent operation makes life difficult for scripters because it means things may possibly get changed out of order or in unanticipated ways, resulting in inconsistencies and errors.

In PowerShell v1, there was no support for the asynchronous pattern, which made it pretty much impossible to handle asynchronous events. To allow for robust handling of asynchronous events, PowerShell v2 added an eventing subsystem that uses a centralized event manager to ensure that this occurs in a rational sequence. This subsystem takes care of all the bookkeeping and synchronization needed to ensure a stable and consistent system without a lot of work on the part of the script author.

#### Note

PowerShell is single threaded, so when it’s busy executing something it can’t handle events. They are queued and executed once PowerShell is available again.

#### Subscriptions, Registrations, and Actions

The scripting model PowerShell uses for handling asynchronous events involves a few core concepts. The first is the idea of an event subscription, where you select the type of events you want to know about and then subscribe to be notified when they occur. These subscriptions are registered with a source identifier, which allows you to give a friendly name to each subscription. Once registered, the event subscription will be notified about relevant events as soon as they occur and will continue to receive notifications until the subscription is cancelled by explicitly unregistering it. Each event subscription may optionally specify an action to be taken.

#### The Eventing Cmdlets

The PowerShell eventing cmdlets, shown in [table 17.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17table01), allow you to register and unregister event subscriptions and list the existing subscriptions. You can also list pending events (as opposed to subscriptions) and handle or remove them as desired. There is also a cmdlet that allows scripts to generate their own events.

| **Table 17.1: The PowerShell eventing cmdlets** | |
| --- | --- |
| **Cmdlet name** | **Description** |
| Register-ObjectEvent | Registers an event subscription for events generated by .NET objects. |
| Register-CimIndicationEvent (Register-WmiEvent) | Registers an event subscription for events generated by WMI objects |
| Register-EngineEvent | Registers an event subscription for events generated by PowerShell itself. |
| Get-EventSubscriber | Gets a list of the registered event subscriptions in the session. |
| Unregister-Event | Removes one or more of the registered event subscriptions. |
| Wait-Event | Waits for an event to occur. This cmdlet can wait for a specific event or any event. It also allows a timeout to be specified limiting how long it will wait for the event. The default is to wait forever. |
| Get-Event | Gets pending unhandled events from the event queue. |
| Remove-Event | Removes a pending event from the event queue. |
| New-Event | Allows the script to add its own events to the event queue. |

When handling events, you need to be able to register actions in response to these events. You do so using cmdlets, but because there are several types or sources of events, there are also several event registration cmdlets, as you saw in the table. The event subscription registration cmdlets are Register-EngineEvent, Register-ObjectEvent, Register-CimIndicationEvent, and Register-WmiEvent. PowerShell-specific events are handled using the Register-EngineEvent cmdlet, asynchronous events on .NET objects are handled using Register-ObjectEvent, and WMI events are addressed with Register-CimIndicationEvent or Register-WmiEvent.

### 17.2.4 Working with Asynchronous .NET Events

You use the Register-ObjectEvent cmdlet to create subscriptions for asynchronous events on .NET objects. First, you need to identify the event you’re interested in. For .NET events, this means that you need an object and the name of the event member on that object to bind. This is the same pattern you’ve already seen with Windows Forms and WPF, where, for example, a Button object has a Click event accessed through the add\_Click() member.

Once you’ve decided on the event to handle, you need to specify what to do with it. The -Action parameter on the cmdlet allows you to provide a scriptblock to execute when an event fires. This scriptblock will receive a lot of information about the event when it’s run, but there may be additional, custom data that you want to pass to the event handler. You can do this with the -MessageData parameter.

Finally, when you’re working with a number of events, the ability to attach a friendly name to the subscription will make things easier to manage. This is what -Source -Identifier is for: It allows you to name the event registration or event source.

There’s one last parameter we haven’t discussed yet: -SupportEvent. In larger event-driven scripts, there may be a number of event registrations that exist only to support higher-level constructs within the application. In these scenarios, it’s useful to be able to hide these supporting events much like the rationale behind the way you hide supporting functions in modules. This event-handler hiding is accomplished using the -SupportEvent switch. As was the case with modules, if you do want to see the hidden events, you can specify the -Force switch on Get-Event-Subscriber.

#### Writing a Timer Event Handler

Okay, enough talk—let’s start doing something with .NET events. One of the most obvious examples of an asynchronous event is a timer. A timer event fires at regular intervals regardless of what else is going on. Let’s see how you can set up subscription events generated by the .NET System.Timers.Timer class.

#### Note

These cmdlets can be used only for asynchronous .NET events. It’s not possible to set up event handlers for synchronous events using the PowerShell eventing cmdlets. That’s because synchronous events all execute on the same thread and the cmdlets expect (require) that the events will happen on another thread. Without the second thread, the PowerShell engine will block the main thread and nothing will ever get executed.

#### Creating the Timer Object

The first thing you need for our example is a Timer object. You use New-Object to create it:

PS> $timer = New-Object -TypeName System.Timers.Timer

Events exist as members on a class, so you can use Get-Member, filtering the results on the Event member type, to see what events this object exposes:

PS> $timer | Get-Member -MemberType Event

TypeName: System.Timers.Timer

Name MemberType Definition

---- ---------- ----------

Disposed Event System.EventHandler Disposed(System.Objec...

Elapsed Event System.Timers.ElapsedEventHandler Elapsed...

From this output, you can see that the Elapsed event is what you’re looking for—it fires when the timer period has elapsed.

#### Setting the Timer Event Parameters

But you need to know more about this object than the events—you need to know how to set the timer interval and start and stop the timer. Again, you can use Get-Member to find this information. (Note that, for brevity, the output shown here has been trimmed to the interesting members.)

PS> $timer | Get-Member

TypeName: System.Timers.Timer

Name MemberType Definition

---- ---------- ----------

Disposed Event System.EventHandler Disp...

Elapsed Event System.Timers.ElapsedEve...

Close Method System.Void Close()

Start Method System.Void Start()

Stop Method System.Void Stop()

ToString Method string ToString()

AutoReset Property System.Boolean AutoReset...

Enabled Property System.Boolean Enabled {...

Interval Property System.Double Interval {...

When you look at the output, the way to start and stop the timer is obvious. The AutoReset property determines if the timer fires only once (AutoReset = $false) or fires repeatedly every interval (AutoReset = $true). Finally, the Interval property controls the firing interval. Because the value is a double, you can guess that it’s specified in milliseconds.

#### Note

Yes, you could’ve gone to the MSDN documentation. But why bother? With Get-Member and a reasonably decent understanding of .NET, Get-Member is often all you need. This makes PowerShell a useful tool for developers as well as IT professionals. Even in Visual Studio, sometimes we’ll still flip over to a PowerShell window to search for information about a type. Simple text and typing is still faster sometimes.

#### Binding the Event Action

Let’s register for an event on this object, which you do with the following command:

PS> Register-ObjectEvent -InputObject $timer `

-EventName Elapsed -Action { Write-Host '<TIMER>' } |

Format-List Id, Name, PSJobTypeName, State, HasMoreData,

Location, Command

Id : 4

Name : d1d302c6-7297-4c0b-b6c7-fc9f02195a2c

PSJobTypeName :

State : NotStarted

HasMoreData : False

Location :

Command : Write-Host '<TIMER>'

This command attaches a scriptblock to the event that will write out the phrase '<TIMER>' when it fires. You have to use Write-Host in this scriptblock because the output from a triggered event action is discarded.

#### Using Register-ObjectEvent

As a handy way to remember how to use the Register-ObjectEvent cmdlet, think of assigning the scriptblock to the event member. If PowerShell supported this, it would look something like this: $timer.Elapsed = { Write-Host "<TIMER>" }.

The Register-ObjectEvent command allows positional parameters in the same order, so the command would look like

PS> Register-ObjectEvent $timer Elapsed { Write-Host "<TIMER2>" }

where the order of the elements is the same: object/member/action.

Now you’ll wait a minute—and nothing happens. That’s because you haven’t done all the other things to the Timer object to make it start firing (though, obviously, binding the event handler beforehand is usually a good idea).

#### Enabling the Event

Let’s complete the remaining steps needed to start the timer triggering. Set the interval to 500 ms so the timer will fire in half a second:

PS> $timer.Interval = 500

You want it to fire repeatedly, so set the AutoReset property to $true:

PS> $timer.AutoReset = $true

Next, enable the timer by setting the Enabled property to $true (or by calling the Start() method, which also sets Enabled to $true):

PS> $timer.Enabled = $true

<TIMER>

<TIMER>

The timer starts running, and you see the output you expected. Next comes the hard part: getting it to stop. The command is easy: type $timer.Stop() and press Enter. But in the console shell, the timer is writing to the screen at the same time you’re typing. This results in scrambled output, looking something like this:

<TIMER>

$timer.Stop()<TIMER>

<TIMER>

(Here’s another place where the ISE works better—the timer output doesn’t interfere with the ability to run commands.) Once you’ve stopped the timer, you can restart it by calling the Start() method a second time:

PS> $timer.Start()

<TIMER>

<TIMER>

<TIMER>

<TIMER>

PS> $timer.Stop()<TIMER>

Now that you know how to register a basic event subscription, we’ll look at how to manage these subscriptions.

#### Managing Event Subscriptions

In this section, you’ll see how to find your event subscriptions and how to remove them when you’ve finished with them. Being able to remove them is important because event subscriptions persist in the session until explicitly removed.

Before you can remove a subscription, you have to find it. PowerShell provides the Get-EventSubscriber cmdlet to do this. Let’s use it to look at the subscription you registered in the previous section:

PS> Get-EventSubscriber

SubscriptionId : 2

SourceObject : System.Timers.Timer

EventName : Elapsed

SourceIdentifier : d1d302c6-7297-4c0b-b6c7-fc9f02195a2c

Action : System.Management.Automation.PSEventJob

HandlerDelegate :

SupportEvent : False

ForwardEvent : False

The Get-EventSubscriber cmdlet returns PSEventSubscriber objects, which have complete information about the registration: the object generating the event, the action to execute, and so on. There are a couple of interesting properties to note in this output.

Because you didn’t give the subscription a friendly name using -SourceIdentifier when you created it, the Register-ObjectEvent generated one for you. This autogenerated name is the string representation of a GUID, so you know it’s unique (but not that friendly). The other thing to notice is that the action shows up as a PowerShell Job object. Because the relationship between events and jobs is a somewhat longer discussion, we’ll defer it to [section 17.2.14](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17lev2sec16).

Now that you can list the event subscriptions, you can set about removing them. You registered event subscriptions with Register-ObjectEvent, so what you need to do is unregister the subscription, which you’ll do with Unregister-Event. The cmdlet noun in this case is Event, not ObjectEvent, because you can use a common mechanism to unregister any kind of event. It’s only the registration part that varies. The rest of the eventing cmdlets remain the same.

When you’re unregistering an event subscription, there are two ways of identifying the event to unregister: by the SubscriptionId property or by the SourceIdentifier. The subscription ID is an integer that’s incremented each time an event subscription is created. Because you didn’t give your event registration a friendly name, you’ll use the SubscriptionId to unregister it:

PS> Unregister-Event -SubscriptionId 2 -Verbose

VERBOSE: Performing the operation "Unsubscribe" on target "Event subscription

'd1d302c6-7297-4c0b-b6c7-fc9f02195a2c'".

Note that you include the -Verbose flag in this command so that you can see something happening. If you try running the command again, it will result in an error. The Unregister-Event cmdlet is silent as long as nothing goes wrong. If something does go wrong, you get an error.

We’ve covered the basics of creating and managing event subscriptions. But before the handlers for these events can do much useful work, they’ll need access to additional information. In the next section, you’ll write more sophisticated handlers and see how they can use the automatic variables provided by the eventing subsystem.

### 17.2.5 Asynchronous Event Handling with Scriptblocks

In this section, we’ll look at the automatic variables and other features that PowerShell provides to allow scriptblocks to be used as effective event handlers.

### 17.2.6 Automatic Variables in the Event Handler

In PowerShell eventing, the scriptblock that handles the event action has access to a number of variables that provide information about the event being handled. These variables are described in [table 17.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17table02).

| **Table 17.2: The automatic variables available in the event handler scriptblock** | |
| --- | --- |
| **Variable** | **Description** |
| $event | This variable contains an object of type System.Management.Automation.PSEventArgs that represents the event being handled. It allows you to access a wide variety of information about the event, as you’ll see in an example. The value of this variable is the same object that the Get-Event cmdlet returns. |
| $eventSubscriber | This variable contains the PSEventSubscriber object that represents the event subscriber of the event being handled. The value of this variable is the same object that the Get-EventSubscriber cmdlet returns. |
| $sender | The value in this variable is the object that generated the event. This variable is a shortcut for $EventArgs.Sender. |
| $sourceEventArgs | Contains objects that represent the arguments of the event being processed. This variable is a shortcut for $Event.SourceArgs. |
| $sourceArgs | Contains the values from $Event.SourceArgs. Like any other scriptblock, if there is a param statement, the parameters defined by that statement will be populated and $args will contain only leftover values for which there were no parameters. |

Let’s write a quick test event handler to see what’s in the object in $Event. You’ll use the timer event again:

PS> $timer = New-Object -TypeName System.Timers.Timer -Property @{

Interval = 1000; Enabled = $true; AutoReset = $false }

In the event subscription action, you’ll display the contents of the event object:

PS> Register-ObjectEvent $timer Elapsed -Action {$Event | Out-Host}

Id Name PSJobTypeName State HasMore

Data

-- ---- ------------- ----- -------

3 54b59faf-5fea-45ff-b086-5c5d3b1eb4c5 NotStarted False

You’ll start the timer to generate the event:

PS> $timer.Start()

ComputerName :

RunspaceId : cd66f2a6-d112-4d09-851d-e02c3f6e459b

EventIdentifier : 1

Sender : System.Timers.Timer

SourceEventArgs : System.Timers.ElapsedEventArgs

SourceArgs : {System.Timers.Timer, System.Timers.ElapsedEventArgs}

SourceIdentifier : 54b59faf-5fea-45ff-b086-5c5d3b1eb4c5

TimeGenerated : 16/05/2017 14:30:01

MessageData :

In this output, you see the properties on the PSEvent object that correspond to the variables listed in [table 17.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17table02). The Timer object that generated the event is available through the Sender property on the object and the $sender variable in the scriptblock. The PSEvent object also includes context data about the event, such as the time the event occurred, the event identifier, and the RunspaceId this event is associated with. The ComputerName property is blank because this is a local event, but in the case of a remote event, it would contain the name of the computer where the event occurred.

### 17.2.7 Dynamic Modules and Event Handler State

Because an event can fire at any time, you might never know what variables were in scope, and this in turn could make it hard to know what state will exist when the action is executed. Instead, you want to be able to run the event handlers in a well-defined, isolated environment. This objective aligns with the design goals for PowerShell modules, so you can leverage this feature by creating a dynamic module ([section 10.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-104-86.xhtml#ch10lev1sec4)) for the action scriptblock. The eventing subsystem does this by calling the NewBoundScriptBlockScriptblock() method to attach a dynamic module to the handler scriptblock.

Beyond ensuring a coherent runtime environment for your event handler scriptblock, the module also allows it to have private state. This ability can be quite useful when you’re monitoring a system’s behavior over time. You can accumulate the information privately and then process it once you’ve gathered enough samples. Let’s look at how this state isolation works. The following is a trivial example where you maintain a count of the number of timer events fired. Once you reach a predetermined limit, the timer will be stopped. Let’s walk through the example. First, you create the Timer object:

PS> $timer = New-Object System.Timers.Timer -Property @{

Interval = 500; AutoReset = $true}

As usual, you subscribe to the Elapsed event on the timer:

PS> Register-ObjectEvent -InputObject $timer `

-MessageData 5 -SourceIdentifier Stateful `

-EventName Elapsed -Action {

$script:counter += 1

Write-Host "Event counter is $counter"

if ($counter -ge $Event.MessageData)

{

Write-Host 'Stopping timer'

$timer.Stop()

}

} > $null

In the handler scriptblock for this event, you’re updating a script-scoped variable $script:counter, which holds the number of times the event has fired. This variable will be visible only within the dynamic module associated with the event, preventing your $counter from colliding with any other users of a variable called $counter. After the variable is incremented, you print the event count and then check to see whether the limit has been reached. Notice that you’re making use of the -MessageData parameter to pass the limit to the event handler, which it retrieves from the MessageData property on the Event object. Now start the timer running to see it in action:

PS> $timer.Start()

Event counter is 1

Event counter is 2

Event counter is 3

Event counter is 4

Event counter is 5

Stopping timer

As intended, the timer message is displayed five times and then the timer is stopped. This example can easily be modified to, for example, monitor CPU usage or process working sets over a period of time.

Setting up action scriptblocks for asynchronous events allows you to efficiently handle events in the background. This, in turn, lets the main thread of your script continue execution in the foreground or, in interactive sessions, allows you to continue entering commands at the shell prompt. There are, however, many monitoring scenarios where there’s no main thread and all you want to do is wait for events to happen. If a service process crashes or faults, you want to be notified so you can take action to restart it. Otherwise, you wait for the next event to arrive. This “wait for an event” pattern is addressed using the Wait-Event cmdlet.

### 17.2.8 Queued Events and the Wait-Event Cmdlet

As an alternative to setting up numerous individual event handler actions, you can use the Wait-Event cmdlet to process events in a loop. This cmdlet allows you to block the PowerShell session, waiting until an event or events happen. When the event arrives, you can take whatever action is required and then loop and wait for the next event. This event loop pattern is common in GUI programming. The syntax for the Wait -Event command is simple:

Wait-Event [[-SourceIdentifier] <string>] [-Timeout <int>]

By using the -SourceIdentifier parameter, you can wait for a specific named event. If you don’t use it, then any unhandled event will unblock you. By using the -Timeout parameter, you can limit the length of time you’ll wait for the event. This allows you to take remedial actions if the event you’re waiting for failed to occur in the prescribed time.

#### Note

You can either register an action for an event or wait for an event, but you can’t do both. If an action has been registered, when the event fires the event object will be removed from the queue and passed to the action scriptblock for processing. As a result, any Wait-Event calls listening for this event will never receive it and will block forever.

Let’s experiment with this cmdlet using something other than the timer event. In this example, you’ll work with the file system watcher class: System.IO.FileSystemWatcher. This class is used to generate events when changes are made to monitored portions of the file system. Let’s look at the events exposed by this type:

PS> [System.IO.FileSystemWatcher].GetEvents() | Select-String .

System.IO.FileSystemEventHandler Changed

System.IO.FileSystemEventHandler Created

System.IO.FileSystemEventHandler Deleted

System.IO.ErrorEventHandler Error

System.IO.RenamedEventHandler Renamed

System.EventHandler Disposed

Using this class, you can register for notifications when a file or directory is created, changed, deleted, or renamed. You can create a FileSystemWatcher object that will monitor changes to your desktop. First, you need to get the resolved path to the desktop folder:

PS> $path = (Resolve-Path ~/desktop).Path

You have to do this because, as discussed previously, when you use PowerShell paths as arguments to .NET methods (including constructors), you must pass in a fully resolved path because .NET doesn’t understand PowerShell’s enhanced notion of paths.

Now, construct the file watcher object for the target path:

PS> $fsw = [System.IO.FileSystemWatcher] $path

Set up an event subscription for the Created and Changed events:

PS> Register-ObjectEvent -InputObject $fsw –EventName Created `

-SourceIdentifier fsw1

PS> Register-ObjectEvent -InputObject $fsw –EventName Changed `

-SourceIdentifier fsw2

Finally, enable event generation by the object:

PS> $fsw.EnableRaisingEvents = $true

At this point, when you call Get-Event, you should see nothing:

PS> Get-Event

This assumes that no other process is writing to the desktop while you’re doing this. Let’s perform an operation that will trigger the event. Create a new file on the desktop:

PS> Get-Date | Out-File -LiteralPath ~/desktop/date.txt

You didn’t set up an action for either of the event registrations, so you won’t see anything happen immediately. The events, however, haven’t been lost. Unhandled events are added to the session event queue where they can be retrieved later. Let’s see what’s in the queue at this point:

PS> Get-Event | select SourceIdentifier

SourceIdentifier

----------------

fsw1

fsw2

In the output, you see that two events have been added: one for the creation of the date.txt file and a second indicating that a change to the containing directory has occurred. Note that reading the events doesn’t remove them from the queue. You need to use the Remove-Event cmdlet to do this—otherwise, you’ll keep rereading the same event objects. The Remove-Event cmdlet allows events to be removed either by SourceIdentifier or by EventIdentifier. To discard all the events in the queue, pipe Get-Event into Remove-Event:

PS> Get-Event | Remove-Event

The queue is now empty, so you can call Wait-Event and the session will block until a new event is generated (or you press Ctrl-C):

PS> Wait-Event

To trigger an event, from another PowerShell session update the date.txt file:

PS> Get-Date > ~/desktop/date.txt

This code will cause an event to be added to the queue, terminating the Wait-Event, which will write the terminating event object to the output stream:

ComputerName :

RunspaceId : cd66f2a6-d112-4d09-851d-e02c3f6e459b

EventIdentifier : 12

Sender : System.IO.FileSystemWatcher

SourceEventArgs : System.IO.FileSystemEventArgs

SourceArgs : {System.IO.FileSystemWatcher, date.txt}

SourceIdentifier : fsw2

TimeGenerated : 16/05/2017 17:07:56

MessageData :

Although you’re unblocked, the event hasn’t technically been handled, so it still exists in the queue and you still have to manually remove it from the queue:

PS> Get-Event | Remove-Event

If you use the -Timeout parameter on Wait-Event and no event is generated, the session will automatically unblock. This makes it easy to distinguish between a timeout and an event.

Now let’s move on to the second type of events that can be handled by the PowerShell eventing infrastructure: CIM events.

### 17.2.9 Working with CIM Events

In this section, we’re going to cover how to work with CIM (WMI) events in PowerShell. As was the case with .NET events, you handle CIM events using a cmdlet to register actions associated with the events: the Register-CimIndicationEvent cmdlet.

#### Note

Register-CimIndicationEvent is a replacement for Register-WmiEvent and should be used in preference to the older WMI cmdlet.

All the other eventing cmdlets remain the same as you saw for object events and will also be the same for any new object sources that might be added in the future.

#### CIM Event Basics

CIM events are, in some ways, considerably more sophisticated than .NET events. First, CIM events are represented as CIM objects and so, like all CIM objects, can be retrieved from either a local or remote computer in a transparent way. Second, because CIM event subscriptions can take the form of a WQL query, event filtering can take place at the event source instead of transmitting all events to the receiver, which is forced to do all the filtering. This is important if you’re monitoring a small set of events on a large number of computers. By doing the filtering at the source (remote) end, far less data is transmitted to the receiver and much less processing needs to be done by the receiver, allowing for the overall monitoring task to scale to far more computers than would otherwise be possible.

#### Note

Unlike object events, there’s no notion of synchronous CIM events, so all event handling must go through the eventing subsystem.

We’ll begin our exploration of CIM events by looking at the Win32\_\*Trace classes, which are much simpler to deal with than the full query-based event subscriptions.

### 17.2.10 Class-Based CIM Event Registration

Before jumping into the full complexity of query-based event subscriptions, we’ll look at some predefined CIM event classes. These classes hide a lot of the complexity required by query-based event registration, making them easier to use. You can use the following command to get a list of these classes—you’ll also display their superclasses to see the relationships between the classes:

PS> Get-CimClass Win32\_\*trace | select CimClassName, CimSuperClassName

CimClassName CimSuperClassName

------------ -----------------

Win32\_SystemTrace \_\_ExtrinsicEvent

Win32\_ProcessTrace Win32\_SystemTrace

Win32\_ProcessStartTrace Win32\_ProcessTrace

Win32\_ProcessStopTrace Win32\_ProcessTrace

Win32\_ThreadTrace Win32\_SystemTrace

Win32\_ThreadStartTrace Win32\_ThreadTrace

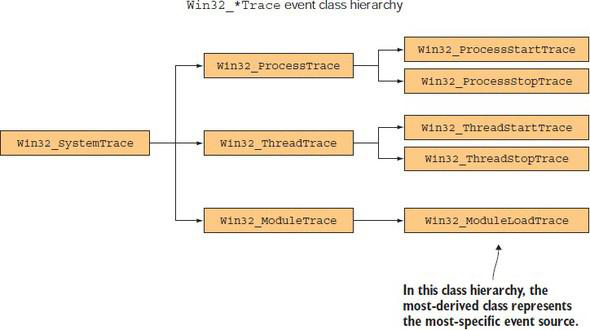
Win32\_ThreadStopTrace Win32\_ThreadTrace

Win32\_ModuleTrace Win32\_SystemTrace

Win32\_ModuleLoadTrace Win32\_ModuleTrace

By inspecting the class/superclass relationships, you can see that these classes form a hierarchy of event sources, where the farther you go from the root, the more specific the event becomes. This hierarchy is illustrated in [figure 17.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17fig04).

Figure 17.4: This figure shows the hierarchy of classes representing simplified WMI event sources. The most-derived class matches the most-specific event. Win32\_ProcessStartTrace will fire only for process starts, whereas Win32\_ProcessTrace will fire for both process starts and process stops



Let’s work through an example that shows how this works.

#### Note

Because these event sources fire for any process event, regardless of who starts them, these commands must be run from an elevated shell on Windows Vista and later. Also, be aware that because you’re recording all process events in the first set of examples, you may see additional output from other processes starting and stopping.

#### Using the Win32\_ProcessTrace Events

You’ll use the Win32\_Process\*Trace classes in this experiment. First, you’ll set up an event subscription to the Win32\_ProcessStartTrace, which will fire every time a process starts:

PS> Register-CimIndicationEvent -ClassName Win32\_ProcessStartTrace -Action {

'Process Start: ' +

$event.SourceEventArgs.NewEvent.ProcessName |

Out-Host

}

You can assign an action scriptblock to these event subscriptions, as you did with object events. In the scriptblock body, you’ll write a message indicating what type of event was fired along with the process name. You’ll set up similar event handlers for the Win32\_ProcessStopTrace and Win32\_ProcessTrace events, again displaying the type of the event and the process name:

PS> Register-CimIndicationEvent -ClassName Win32\_ProcessStopTrace -Action {

'Process Stop: ' +

$event.SourceEventArgs.NewEvent.ProcessName |

Out-Host

}

PS> Register-CimIndicationEvent -ClassName Win32\_ProcessTrace -Action {

'Process Any: ' +

$event.SourceEventArgs.NewEvent.ProcessName |

Out-Host

}

From the hierarchy (and the names of the events), you know that Win32\_ProcessStartTrace fires when a process starts, Win32\_ProcessStopTrace fires when a process is terminated, and Win32\_ProcessTrace fires on either kind of process event. To test these subscriptions, run the following command, which will start and stop an instance of the calc process a number of times (on Windows 10 – earlier versions of Windows called the process calc:

PS> & {

Start-Process calc

Start-Sleep 3

Stop-Process -Name Calculator

Start-Sleep 3

Start-Process calc

Start-Sleep 3

Stop-Process -Name Calculator

Start-Sleep 3

}

In this command, you’re using Start-Process to start the calc process. After three seconds, you use Stop-Process to terminate the calculator instance. This pattern is repeated two times, and the whole thing is wrapped in a scriptblock to cause it to be executed as a single command, so you avoid having your commands mixed in with the output and cluttering things up. Here’s the output produced by this command (Windows 10 also refers to calc.exe as calculator.exe):

Process Start: calc.exe

Process Start: Calculator.exe

Process Any: calc.exe

Process Any: Calculator.exe

Process Any: calc.exe

Process Stop: calc.exe

Process Stop: Calculator.exe

Process Any: Calculator.exe

Process Start: calc.exe

Process Any: calc.exe

Process Start: Calculator.exe

Process Any: Calculator.exe

Process Any: calc.exe

Process Stop: calc.exe

Process Stop: Calculator.exe

Process Any: Calculator.exe

The first two records were generated by the first calc process starting. You get both Win32\_ProcessStartTrace and Win32\_ProcessTrace firing, but not Win32\_ProcessStopTrace. The calc process is then stopped, resulting in two more records, and this is repeated one more time for a total of eight records. (The order in which the specific and general events are fired is nondeterministic, so the exact order will change with different runs of the start/stop command.)

The final step in this experiment is to clean up the event subscriptions you created. Here’s the easiest way to do that:

PS> Get-EventSubscriber | Unregister-Event

PS> Get-Job | Remove-Job

#### Note

This code removes all event subscriptions for this session. That’s fine for experimentation, but you should be careful doing this in a production environment and be selective about what is removed.

This completes the easy part of CIM event handling. Although setting up event handlers this way was easy, it was also limited. When you retrieve CIM object instances using Get-CimInstance, you’re able to perform sophisticated filtering and can be precise about the objects you retrieve. You can be as precise with events, but doing so requires the use of WQL queries. We’ll cover that in the next section.

#### Query-Based CIM Event Registrations

In [chapter 16](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-16-125.xhtml#ch16), you used the WMI Query Language to select and filter CIM objects. The format of those instance-based WQL queries was

SELECT <propertyList> FROM <ObjectClass> WHERE <predicateExpression>

With a little bit of additional syntax, WQL can also be used to select and filter CIM events.

#### Note

In CIM parlance, what you filter is called a notification, not an event. CIM defines an event as something that happens at a particular time like a process starting or a user logging on. Notifications are the object representation (or model) for these event occurrences. For simplicity, we’re going to stick to using event for both cases in the rest of this chapter.

The core syntax for event queries is the same as for instance queries but with some additional features. We’ll look at these features in the next couple of sections.

#### The WITHIN Keyword

The first of the additional keywords we’ll discuss is WITHIN. This keyword is used in a query as follows:

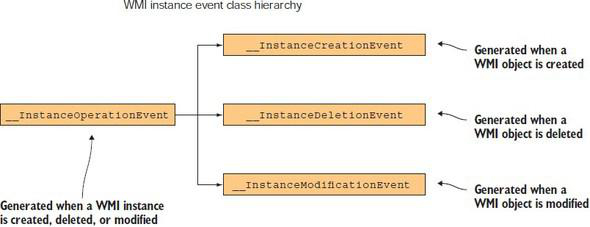
SELECT <propertyList> FROM <EventClass> WITHIN <Seconds> WHERE <predicateExpression>

The WITHIN keyword is used to specify the polling interval that the WMI service should use to monitor and relay event data. The polling interval is the frequency with which the monitored resource is checked. The smaller the polling interval, the more often the monitored resource will be checked. This results in faster and more accurate event notifications, but it also places a greater burden on the monitored system. The argument to the WITHIN keyword is a floating-point number. This means you could theoretically specify polling intervals of less than one second. But specifying a value that’s too small (like 0.001 seconds) could cause the WMI service to reject a query as not valid due to the resource-intensive nature of polling. The polling interval should be chosen based on the type of event being monitored. If the event doesn’t require instant action, it’s generally recommended that the polling interval be greater than 300 seconds (5 minutes).

#### The CIM Intrinsic Event Classes

The objects you query for are also a bit different. With object events, you create an instance of an object and then subscribe to an event on that object. With CIM event queries, you subscribe to the type of event and then specify the event-generating class you’re interested in. Some of the most useful of these intrinsic-event classes are \_Instance-CreationEvent, \_InstanceDeletionEvent, and \_InstanceModificationEvent, which are all derived from \_InstanceOperationEvent. These classes and their relationships are shown in [figure 17.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17fig05).

Figure 17.5: The class hierarchy for the CIM instance operation event class. These events are generated when a CIM is object is created, deleted, or modified. The base event class is triggered for all three



These classes mirror the pattern you saw in the previous section, where Win32\_ProcessTrace was the root event with Win32\_ProcessStartTrace and Win32\_ProcessStopTrace as derived events. The difference here is that there’s no class like Win32\_Process mentioned in these events. They are general-purpose events generated by all objects. When you want to register an event subscription for one of these events, you use the ISA operator to select which class you’re interested in receiving instance notifications from. Let’s see what a query using the WITHIN keyword and these instance notifications events looks like:

PS> $svcQuery = @"

SELECT \* FROM \_\_InstanceOperationEvent WITHIN 1

WHERE TargetInstance ISA 'Win32\_Service'

AND TargetInstance.Name='BITS'

"@

This query says to retrieve all events from InstanceOperationEvent with a polling interval of 1 second (this is an experiment, so you use a small value) where the object generating the event is an instance of the Win32\_Service class and the Name property on the instance is BITS (Background Intelligent Transfer Service). You want to generate an event anytime something happens to the BITS service.

Use the Register-CimIndicationEvent cmdlet to subscribe to this event. In the action field, display a message indicating the source of the event and then print out the contents of the $event variable:

PS> Register-CimIndicationEvent -Query $svcQuery -Action {

Write-Host 'Got instance operation event on Win32\_Service'

$Event | Format-List \* | Out-Host

}

With the event subscription set up, trigger the event by starting the BITS service:

PS> Start-Service BITS

Got instance operation event on Win32\_Service

ComputerName :

RunspaceId : 2e7fa8de-aa03-4061-bce7-edf3a58d846d

EventIdentifier : 1

Sender : Microsoft.Management.Infrastructure.CimCmdlets.CimIndicationWatcher

SourceEventArgs : Microsoft.Management.Infrastructure.CimCmdlets.CimIndicationEventInstanceEventArgs

SourceArgs : {Microsoft.Management.Infrastructure.CimCmdlets.CimIndicationWatcher, }

SourceIdentifier : 2587f4cc-fea5-4711-8a6d-4ae648e2524d

TimeGenerated : 16/05/2017 17:20:20

MessageData :

After a second or so, you see the message printed out by the action scriptblock. Stop the service and you get a second message because the event you’ve subscribed to fires for any change.

In the next section, we’ll look at additional features for improving the network behavior of the system by grouping events instead of sending them one at a time.

#### Aggregating Events with GROUP

The next keyword we’ll cover is GROUP. The GROUP clause is used to aggregate the events based on certain criteria. This means that instead of generating one notification per event, the WMI service will group them together with a count and a representative instance. This is another way to reduce the load on the client and the network:

SELECT \* FROM EventClass [WHERE property = value]

GROUP WITHIN interval [BY property\_list]

[HAVING NumberOfEvents operator integer]

You create a query-based WMI event registration using the -Query parameter set on Register-CimIndicationEvent. Let’s set up this new event subscription. First, save your query in a string and set up a counter that will record the total number of events:

PS> $GroupQuery = @"

Select \* From \_\_InstanceOperationEvent Within .5

Where TargetInstance Isa 'Win32\_Service'

and TargetInstance.Name='BITS'

Group Within 20

"@

PS> $global:TotalEvents = 0

Now register this event subscription:

PS> Register-CimIndicationEvent -Query $GroupQuery -Action {

Write-Host 'Got grouped event'

$ne = $Event.SourceEventArgs.NewEvent

$ti = $ne.Representative.TargetInstance

$global:TotalEvents += $ne.NumberOfEvents

$msg = 'Type: ' + $ne.\_\_CLASS +

' Num Evnts: ' + $ne.NumberOfEvents +

' Name: ' + $ti.Name +

' (' + $ti.DisplayName + ')' |

Out-Host

}

In the body of the event action scriptblock, you’ll format a string containing some of the more interesting fields (at least for the purpose of this experiment). You’ll show the type of the event class, the number of events that have been aggregated, and then the Name and DisplayName for the matched service. You’ll generate a series of events using a foreach loop to cause the event aggregation to fire:

PS> foreach ($i in 1..3){

Start-Service -Name BITS

Start-Sleep 2

Stop-Service -Name BITS

Start-Sleep 2

}

These events will all be accumulated in the event group, and when the group interval expires, you should get an event notification. Use the Start-Sleep command to wait for the timeout to expire:

PS> Start-Sleep 10

Got grouped event

Type: Num Evnts: 6 Name: BITS (Background Intelligent Transfer Service)

The event count shows your total:

PS> "Total events: $TotalEvents"

Total events: 6

Now that you have your event, let’s clean up the event subscription:

PS> Get-EventSubscriber | Unregister-Event

In this example, you’ve seen how you can use the GROUP keyword to further reduce the number of events that need to be sent to the monitoring script.

This completes our look at CIM eventing, so let’s move on to something a bit different. Up until now, we’ve only been talking about how to respond to events. In the next section, you’ll see how to generate some events of your own.

### 17.2.11 Engine Events

The last category of events we’re going to look at is engine events. With engine events, the notifications are generated by the PowerShell engine itself, either through one of the predefined engine events or by explicitly generating an event in a script using the New-Event cmdlet.

#### Predefined Engine Events

There’s currently only one predefined engine event identified by the string “PowerShell.Exiting”. This string can also be retrieved using a static method as follows:

PS> [System.Management.Automation.PsEngineEvent]::Exiting

PowerShell.Exiting

This event is triggered when the PowerShell engine is shutting down and allows you to perform actions before the session exits. Here’s an example event registration:

PS> Register-EngineEvent `

-SourceIdentifier PowerShell.Exiting `

-Action {

"@{Directory='$PWD'}" > ~/pshState.ps1

}

This command registers an action to take when the PowerShell session ends. This action writes a hashtable to the file pshState.ps1 in the user’s directory. The hashtable captures the user’s current directory at the time the session was exited. Let’s use this in an example. You’ll create a child PowerShell.exe process to run your script so you don’t have to exit the current process. PowerShell recognizes when a scriptblock is passed to the PowerShell.exe command and makes sure that everything gets passed to the command correctly. Let’s run the command:

PS> powershell {

Register-EngineEvent `

-SourceIdentifier PowerShell.Exiting `

-Action {

"@{Directory='$PWD'}" > ~/pshState.ps1

} | Format-List Id,Name

cd ~/desktop

exit

}

Id : 3

Name : PowerShell.Exiting

Now look at the content of the file:

PS> Get-Content ~/pshState.ps1

@{Directory='C:\Users\brucepay.REDMOND\desktop'}

You see that the file contains a hashtable with the current directory recorded in it. This example can easily be expanded to include things like the user’s history or the contents of the function: drive, but adding those extensions is left as an exercise for the reader.

The other class of engine events is script-generated events. We’ll look at those next.

### 17.2.12 Generating Events in Functions and Scripts

The last of the core eventing cmdlets to look at is the New-Event cmdlet. This cmdlet allows a script to generate its own events. Let’s use this cmdlet in an example to see how it works. First, you create the timer object:

PS> $timer = New-Object System.Timers.Timer -Property @{

Interval = 5000; Enabled = $true; AutoReset = $false }

Then you register the event subscription:

PS> Register-ObjectEvent $timer Elapsed -Action {

Write-Host '<TIMER>'

New-Event -SourceIdentifier generatedEvent -Sender 3.14

} > $null

In the handler scriptblock, as well as writing out a message, you’re calling New-Event to generate a new event in the event queue. Finally, start the timer

PS> $timer.Start() > $null

and wait for the event. Pipe the object returned from Wait-Event into the foreach cmdlet for processing:

PS> Wait-Event -SourceIdentifier generatedEvent |

foreach {

'Received generated event'

$\_ |

Format-Table -AutoSize SourceIdentifier, EventIdentifier, Sender

$\_ | Remove-Event

}

Received generated event

SourceIdentifier EventIdentifier Sender

---------------- --------------- ------

generatedEvent 2 3.14

You see the output from Wait-Event. In the foreach block, you display the source identifier of the event generated by New-Event, and the Sender field shows the number you passed to the cmdlet. When you’ve finished with this example, you’ll remove the event subscription:

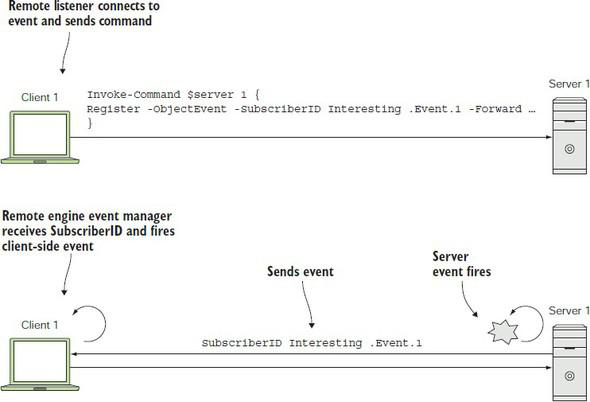
PS> Get-EventSubscriber | Unregister-Event

This pretty much completes the local event-handling story. But with PowerShell’s remoting capabilities, obviously your eventing infrastructure needs to work in a distributed environment as well. In the next section you’ll see how to work with events in remote scenarios.

### 17.2.13 Remoting and Event Forwarding

Being able to set up local event handlers is useful, but you also need to be able to process events generated on remote computers to manage distributed datacenters. The PowerShell eventing subsystem, by building on top of PowerShell remoting, makes this surprisingly easy. In [figure 17.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17fig06) notice the -Forward parameter. This parameter does exactly what you might expect: it forwards the subscribed event to a remote session. This is where the -SourceIdentifier parameter becomes critical. The source identifier name that’s specified at the event source end becomes the name of the event to process on the receiving end. This process is illustrated in [figure 17.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17fig06).

Figure 17.6: The second-hop authentication changes when credential delegation is used. Without delegation, the second hop from server 1 to server 2 authenticates as the user that the service is running under. With credential forwarding enabled, server 1 can use the client credentials to authenticate to server 2 as the client user



Here’s where the engine events come into play. The forwarded events are handled using engine event processing. Register-EngineEvent lets you register subscriptions that trigger the event handler based on the subscription identifier sent from the remote end. The events generated by New-Event in the previous section are also engine events. In the next section, we’ll look at a detailed example where you forward an event from one machine for processing on another.

#### Handling Remote EventLog Events

In this section, you’re going to apply what you’ve learned. Your goal is to be notified locally every time an event is written into the event log on a remote computer. The .NET EventLog class exposes such an event: EntryWritten. To set this up, you must establish event forwarding on the remote machine and then register a load event handler. You’ll also need to maintain a connection to the remote end using the duration of time you want to get events because the events are being forwarded over this channel.

The first thing you need to do is to establish a connection to the target computer. You do so with the New-PSSession cmdlet, passing credentials if needed:

PS> $s = New-PSSession -ComputerName W16DSC01

This is the session you’ll use to set up the event forwarding and then transfer the forwarded events. Next, you’ll use Invoke-Command to set up the event-forwarding registration. The code to do that looks like this:

PS> Invoke-Command -Session $s {

$myLog = New-Object System.Diagnostics.EventLog application

Register-ObjectEvent `

-InputObject $myLog `

-SourceIdentifier EventWatcher1 `

-EventName EntryWritten `

-Forward

$myLog.EnableRaisingEvents = $true

}

Inside the scriptblock passed to Invoke-Command, you’re creating an EventLog object associated with the Application event log. Then you use Register-ObjectEvent to set up event forwarding for events that occur on the EntryWritten event. You’ll use the source identifier name EventWatcher1. Finally, you enable raising events on the event log object.

With the remote end configured, it’s time to set up the local end. This task is much simpler. You register an engine event handler that will trigger on the source ID matching the remote end:

PS> Register-EngineEvent -SourceIdentifier EventWatcher1 -Action {

param($sender, $event)

Write-Host "Got an event: $($event.entry.message)"

}

And you’re finished. Now whenever an entry is added to the Application event log on the remote computer, you’ll see the entry message displayed on your console. If you’re impatient, you can trigger an event yourself. Use the .NET FailFast() API to cause a “Watson” event to be generated by crashing a PowerShell process on the remote machine:

PS> powershell "[System.Environment]::FailFast('An event')"

After a short time, you’ll see something like the following displayed on the console:

Got an event:

Well, this sort of worked. The event did trigger the event handler, and you got the part of the event you wrote. Unfortunately, the most interesting piece—the message in the event itself—is mysteriously absent. You’ll see what happened in the next section.

#### Serialization Issues with Remote Events

The serialization mechanism used by remoting can sometimes cause problems when using remote events. Because the event is being sent over the remoting channel, it has to be serialized by the PowerShell serializer. By default, the serialization depth is only 1. This means you get the top-level properties but not the second-level properties. To preserve the message content in $event.Entry.Message, you need to change the serialization depth for this type of object to 2. You need an XML document that you can pass to Update-TypeData to change the serialization depth for System.Diagnostics.EntryWrittenEventArgs to 2. Save this XML in a variable as a string for now:

$typeSpec = @'

<Types>

<Type>

<Name>System.Diagnostics.EntryWrittenEventArgs</Name>

<Members>

<MemberSet>

<Name>PSStandardMembers</Name>

<Members>

<NoteProperty>

<Name>SerializationDepth</Name>

<Value>2</Value>

</NoteProperty>

</Members>

</MemberSet>

</Members>

</Type>

</Types>

'@

Now before you use this to set up new events, you should remove the existing event registrations on both the local and remote ends of the connection:

PS> Invoke-Command $s { Unregister-Event EventWatcher1 }

PS> Unregister-Event EventWatcher1

You have the XML in a local variable but you need to update the type metadata on the remote end. You need to get the content of the $typeSpec variable over to the remote machine, which you’ll do by passing it as an argument to the Invoke-Command scriptblock:

PS> Invoke-Command -ArgumentList $typeSpec -Session $s {

param ($typeSpec)

$tfile = New-TemporaryFile

$newfilename = $tfile.FullName -replace '\.tmp$', '.ps1xml'

Rename-Item -Path $tfile.FullName -NewName $newfilename

Set-Content -Value $typeSpec -Path $newfilename

Update-TypeData -PrependPath $newfilename

Remove-Item -Path $newfilename -Force

}

Let’s go over what’s happening in this scriptblock. First, you’re using the PowerShell v5 cmdlet New-TemporaryFile to create a temporary file in your TEMP folder. Because the default extension on the filename that’s returned is .tmp and you need it to be .ps1xml, you use the -replace operator to change the extension and rename the file. Then you write $typeSpec to the file using Set-Content, call Update-TypeData to load the file, and clean up by removing the temp file.

With the type metadata updated, you can set up the remote event registration as before:

PS> Invoke-Command $s {

$myLog = New-Object System.Diagnostics.EventLog application

Register-ObjectEvent `

-InputObject $myLog `

-SourceIdentifier EventWatcher1 `

-EventName EntryWritten `

-Forward

$myLog.EnableRaisingEvents = $true

}

then set up the local event subscription:

PS> Register-EngineEvent -SourceIdentifier EventWatcher1 -Action {

param($sender, $event)

Write-Host "Got an event: $($event.entry.message)"

}

And finally, you’re ready to try your event trigger on the remote machine again:

PS> powershell "[System.Environment]::FailFast('An event')"

This time, you’ll see the event messages including the text from the call to FailFast() as written into the event log on the remote system.

Congratulations! We’ve pretty much reached the end of our eventing discussion and you’re still with us. Event processing is an advanced topic, even for full-time programmers. Understanding how multiple actions are going to interoperate can be mind-boggling. PowerShell’s approach to eventing is designed to make this as simple as possible, but understanding how it works under the hood can go a long way toward helping you figure things out. Let’s take a peek.

### 17.2.14 How Eventing Works

The eventing infrastructure relies on two other components of PowerShell: modules (for isolation, as discussed earlier) and jobs (for managing subscriptions). When you registered an event subscription, you saw that an object was returned. This object is, in fact, a job object, with the same base class as the object you get back from Start-Job or the -AsJob parameter on Invoke-Command. Once an event subscription is created, it will show up in the Job table, which means you can use the Get-Job cmdlet as another way to find this subscription. Let’s go back to our timer event subscription and see what this looks like:

PS> $timer = New-Object -TypeName System.Timers.Timer

PS> Register-ObjectEvent -InputObject $timer `

-EventName Elapsed -Action { Write-Host '<TIMER>' }

PS> Get-Job | Format-List

Module : \_\_DynamicModule\_c83413eb-bad9-47eb-88b0-e4d38ff2aa7f

StatusMessage :

HasMoreData : False

Location :

Command : Write-Host '<TIMER>'

JobStateInfo : NotStarted

Finished : System.Threading.ManualResetEvent

InstanceId : 1f73bb6b-5fe0-4ce4-8d2e-f750f3a4c1ed

Id : 4

Name : d49bc9da-dfd5-4b5a-9cc9-5b44b508415c

ChildJobs : {}

PSBeginTime :

PSEndTime :

PSJobTypeName :

Output : {}

Error : {}

Progress : {}

Verbose : {}

Debug : {}

Warning : {}

Information : {}

State : NotStarted

Let’s start the timer running again, setting the interval to something large so you can still type:

PS> $timer.Interval = 60000

PS> $timer.Start()

Now when you run Get-Job after the timer has started (you may need to wait a little while)

PS> Get-Job | Format-Table State,Command -AutoSize

State Command

----- -------

Running Write-Host '<TIMER>'

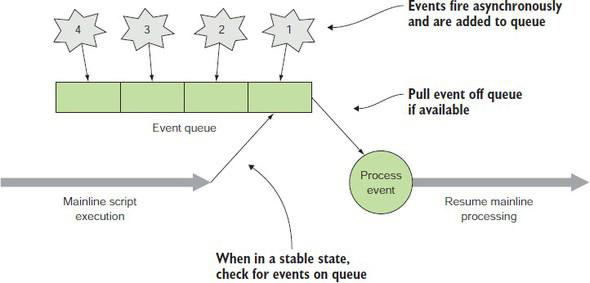
you see that the job state has been changed to Running. The other thing you should be able to do if it’s a Job is to stop it by calling Stop-Job. It works. But this code has done more than stop the job—it’s also removed the event subscription!

Because event handlers are effectively running in the background, it seems logical to manage an active subscription as a Job. You should note that, although the executing event handler is represented as a Job, it wasn’t started using Start-Job and, unlike PowerShell jobs, still runs in process with the session that set up the subscription.

At the beginning of our discussion on events, we talked about the issues involved in dealing with asynchronous events. Because these events can occur in any order, great care is required to make sure that the integrity of shared data structures is maintained. To maintain this integrity, you have to make sure that programs synchronize access to the shared objects, and doing so turns out to be difficult. In fact, this is one of the most common reasons that a program stops responding and appears to be hung. If two actions are trying to update a synchronized object at the same time, they can end up blocking each other, each trying to get exclusive access to the resource. This type of contention is called a deadlock.

PowerShell deals with this problem by imposing a strict order on the actions instead of on individual data objects. When an asynchronous event occurs, the eventing subsystem adds that event object to the event queue. Then, at various points in the PowerShell runtime, the engine checks to see if there are any events posted to the event queue. If there are, the engine suspends the mainline activity, pulls an event off the queue, switches to the module context for that event handler, and then executes the event scriptblock. This queuing mechanism is illustrated in [figure 17.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-172-133.xhtml#ch17fig07).

Figure 17.7: How asynchronous event processing is handled in PowerShell. As events occur, they’re added to the queue asynchronously. At various stable points, the engine checks the queue and pulls events off to execute. Once the event execution is complete, normal processing resumes



Events are added to the queue as they arrive and then are pulled off the queue by the engine and processed when a convenient spot is reached.

To make sure events are processed in a timely manner, the engine needs to check the queue fairly often, but if it checks too often, it will substantially slow down the interpreter. In PowerShell, the engine checks for events in all calls that write objects, including between each stage in a pipeline. It also checks between each statement in a script and anywhere the engine might loop for a long time. This provides a good trade-off between event latency and overall performance. In the case where multiple events are pending on the queue at the time of the check, the engine will use a throttling policy to decide how many of the pending events will be processed before returning to the mainline so that the foreground activity isn’t “starved.” (As an aside, the places where the event queue is checked are the same places that the engine checks to see if it has been requested to stop executing, such as when the user presses Ctrl-C.)

If the event has an action block associated with it, that scriptblock executes until it’s completed. Once the event action is finished, the mainline activity is resumed. Because the engine processes events only when it knows the system state is stable, problems related to inconsistent system state don’t arise, and all activity is effectively synchronous.

#### Note

An event action runs until it’s complete. As long as it’s running, no other events are processed, and the mainline activity is suspended. This means that event handlers shouldn’t be written to execute for a long time. The same consideration exists when writing GUIs. If a control’s event handler runs on the UI thread for a long time, the UI will be blocked, unable to respond to events, causing it to appear to hang.

This architecture isn’t as efficient as the more fine-grained techniques, so it’s not appropriate for programs that are performance-sensitive. It is, however, simple, effective, and completely sufficient for PowerShell scripting. It makes asynchronous event handling in PowerShell a reasonable if somewhat advanced proposition.

## 17.3 Summary

* PowerShell doesn’t load all .NET classes by default.
* Use Add-Type to load additional assemblies.
* You can write GUI applications in PowerShell, but that doesn’t mean you should.
* PowerShell can work with WinForms or WPF.
* PowerShell can work with events from .NET, CIM, and the PowerShell engine.
* Synchronous events occur one at a time—for instance, a button click in a GUI application.
* Asynchronous events can occur at any time and can interrupt other actions.
* Events are registered in a PowerShell session. If the session is closed, the registrations are lost.
* Use a source identifier to identify events.
* Use scriptblocks to define the action to be taken if an event is triggered.
* New-Event is used to create events from within scripts and functions.
* Events work with the job system.
* Events can be forwarded from a remote machine to the local machine.
* Serialization can cause problems with remote events—you need to change serialization depth to 2.

It’s time to investigate some of the newer features in PowerShell. We’ll start with Desired State Configuration in the next chapter.

## Chapter 18: Desired State Configuration

### Overview

This chapter covers

* The need for Desired State Configuration (DSC)
* Configuration management theory
* DSC architecture
* DSC modes: push and pull
* Local configuration manager
* Partial configurations

*Make it so!  
Captain Jean-Luc Picard, USS Enterprise-D*

The need to build and configure servers quickly in a consistent, repeatable manner has been a longstanding problem in IT. One solution is to adopt Desired State Configuration (DSC), a PowerShell extension introduced with Windows Server 2012 R2 (PowerShell v4) and extended in Windows Server 2016 (PowerShell v5). DSC provides a mechanism to manage the configuration of your server estate, including:

* Add or remove Windows features
* Manage registry, files, and folders
* Manage processes and services
* Install and manage software packages

In addition, DSC can monitor the server configuration you’ve applied and, if necessary, reset the configuration to the desired state if the current configuration has been modified so that it doesn’t match the desired state. DSC can also be configured to manage reboots required by configuration changes.

In this chapter, we’ll introduce you to DSC. We’ll start by explaining the need for DSC and going over the underlying theory of configuration management. After reviewing the architecture of DSC, we’ll demonstrate how it works with examples.

DSC can work in two modes: push and pull. In push mode, you’re responsible for delivering the configuration to the server. In pull mode, the target server pulls its configuration from the DSC server. We’ll explain how to set up both options and why push mode scales better to the enterprise.

We’ll also explore the Local Configuration Manager, a DSC component local to each target server. The chapter closes with a look at how configurations can be broken into parts that can be managed by different teams, called partial configurations.

First let’s look at the theory.

## 18.1 DSC Model and Architecture

In this section, we’ll look at why you need DSC, then examine the DSC model and architecture. First, let’s recap why we need configuration management in general and DSC in particular.

### 18.1.1 The Need for Configuration Management

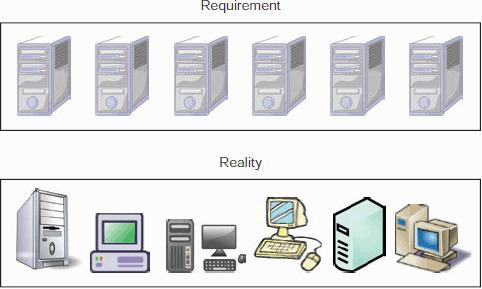
The number of servers in an organization has grown significantly over the years, and continues to grow. The introduction of virtualization, containers, and programming methodologies such as Agile programming, means that servers can and must be created quickly to meet the changing business needs of the organization. Organizations are moving new applications into production with increasing frequency; sometimes multiple new builds are issued in a day, all needing new servers.

The “traditional” method of manually configuring servers fails for a number of reasons:

* **It’s a slow process—** Installing and configuring the operating system and required software can take a minimum of several hours.
* **The process is error prone—** Even with checklists it’s easy to miss a step.
* **It’s non-repeatable—** You can’t guarantee that two servers will be configured identically. Different administrators may have differing views about how a server should be configured or be working from different versions of the build instructions.
* **Undocumented changes are made—** Configurations drift from the baseline with time.

This situation is also summed up in [figure 18.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-181-136.xhtml#ch18fig01), where the requirement is to build six identical servers. In reality, even if the same person builds each server, there will be differences. The potential differences become greater if the servers are built by different people.

Figure 18.1: A common requirement of six identically configured servers. The reality is that each will be different



One approach is to script your builds. Scripting has a number of drawbacks, including the fact that not all administrators are comfortable with it. Also, ensuring that a common version of the script is used can be problematic. Some configuration tasks require advanced scripting skills that may not be available in the organization.

The server configuration problem becomes twofold: first, you need the processes in place to manage configuration management, and second, you need the tools to perform configuration management. Configuration management is part of a wider DevOps framework and should be introduced into your organization as part of your adoption of DevOps processes. How you move your organization to embracing the DevOps principles is outside of our scope.

A number of tools have been created in recent years for managing configurations. Some examples are Puppet ([https://puppet.com](https://puppet.com/)), Chef ([www.chef.io](http://www.chef.io/)), Salt ([https://saltstack.com](https://saltstack.com/)), and Ansible ([www.ansible.com](http://www.ansible.com/)). These tools are all from the UNIX/Linux world and so require a Linux system to install them on. As a consequence of their origin, Puppet, Chef, and the other tools work well in the Linux space. But when it came to implementing their toolsets on Windows, they’ve struggled. This was partly because Windows is an API-driven operating system, as opposed to the document-driven nature of Linux. Also, many Windows administrators have been reluctant to learn Linux merely to bring configuration management into their environment.

Enter DSC. It is intended to supply a basic configuration management framework for Windows that can be used directly by Windows administrators. It’s also intended to make it easier for the manufacturers of existing configuration tools to work with Windows by enabling them to use DSC.

Now you know why you need DSC. Before we look at its architecture, let’s look at the DSC model of configuration management.

### 18.1.2 Desired State Configuration Model

Creating a PowerShell script to configure one or more servers builds on the knowledge and skills you already possess. Everything you’ve learned in the book so far can help you create those scripts.

DSC introduces you to a different way of thinking. You’re telling the system how you want it to be configured, you’re not necessarily worried about how it gets to the desired state. You are in effect creating a model of the desired state and applying that model. In practice, this means that you create a configuration that is transported and applied to the target server. The sequence of operations is:

1. A configuration is created.
2. A MOF file is generated from the configuration.
3. The MOF file is transported to the target server.
4. The target server implements the configuration.

#### MOF files

DSC uses Managed Object Format (MOF) files to transfer configuration information to the target machine. MOF is part of the DMTF CIM standard (originally implemented on Windows as WMI—see [chapter 16](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-16-125.xhtml#ch16)). The MOF file generated from a configuration is a fully resolved pure-data representation of the configuration. There are no unresolved variables in the MOF file. MOF was chosen because it’s a format that allows you to represent the classes defining an object as well as instances of those objects.

In this section, we’ll discuss these concepts and explain how DSC differs from the PowerShell scripting you know and love.

#### Declarative Programming

One of the exercises from our English lessons in school was having to write a set of instructions to perform a task. Think about making a cheese sandwich. You have to perform a number of discrete steps:

* Remove two slices of bread from packet
* Butter one side of each slice
* Put one slice on plate butter side up
* Cut cheese into slices
* Place slices on bead
* Put second slice on top of cheese, butter side down
* Cut sandwich in half

That list shows the major steps. The process could easily run to 30 or more steps if each were broken down further. You may not realize it, but this is how your PowerShell scripts work—you provide a set of instructions that PowerShell (hopefully) executes to completion. This approach is known as procedural, or imperative, programming. You tell the system how to perform the tasks.

DSC doesn’t work this way. DSC is declarative. You tell the system how you want it to be configured, and it goes off and performs the task without your having to provide all the intermediate commands. Captain Picard doesn’t tell his subordinates how to do things. He tells them what they have to do, and when they’re ready to proceed he says, “Make it so.” Applying that philosophy to our cheese sandwich example, we’d have a single step that stated, “Make a cheese sandwich.”

Now that you understand how you’ll be thinking about things in a different way, let’s look at the DSC model.

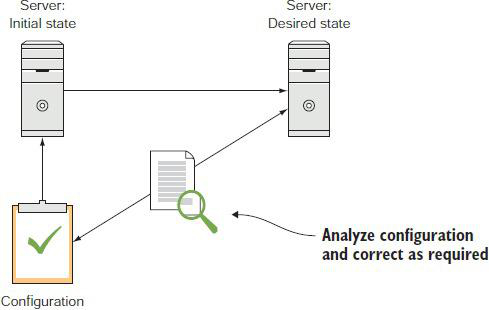
#### DSC Model Breakdown

The DSC model requires three things:

1. An external representation of the desired state of the system, called a configuration
2. A way to get and set the system state
3. A way to compare the desired state against the current state and enact the changes that need to be made to bring the system into compliance with the desired state

This model is illustrated in [figure 18.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-181-136.xhtml#ch18fig02).

Figure 18.2: DSC model showing the initial and desired state



Let’s examine the individual model parts. We’ll be covering each in greater depth later in the chapter. The most important point is that you need a representation of the desired state that exists independently from the current state of the system. This allows you to compare the desired state against the current state, compute the differences, and then perform the necessary steps to bring the current state into compliance with the desired state.

#### DSC Configuration

The representation of the desired state of the system is called a configuration. In DSC, a configuration is made up of components called resources, which represent concrete aspects of the system, like files, processes, or services. An example of a DSC configuration containing a single resource looks like this:

Configuration AddFile {

File TestFolder {

Ensure = 'Present'

Type = 'Directory'

DestinationPath = 'C:\TestFolder'

Force = $true

}

}

The configuration checks whether a folder called TestFolder is present on the C: drive of the target machine. If the folder isn’t found, it is created. We’ll cover creation of DSC configurations in more detail later in this chapter.

Once you have a configuration, you need a way to apply it against the target machine. Doing that involves testing the current configuration and making any changes to bring it in line with the desired configuration.

#### Note

DSC requires the ability to uniquely identify a resource on the system: the key property. (In some cases, such as the WindowsProcess resource, this had to be fudged by the PowerShell team).

A DSC configuration represents a single terminal state for the target machine. That’s why you can’t have a document that says a resource is both 1 and 0, because that’s a temporally impossible terminal state.

The DSC agent is monotonic in operation—each resource moves you closer to the desired terminal state. It never moves you farther away (though the resource implementation may do that internally).

#### DSC Resources

DSC configurations are created using DSC resources. The previously mentioned configuration uses the File resource, though we didn’t explicitly state this. The File resource is one of a small number of DSC resources installed with PowerShell. These resources enable you to

* Manage files and folders
* Manage the registry, event logs, processes, and services
* Manage Windows operating system roles and features
* Manage local users and groups

There are many more resources on the PowerShell Gallery—over 900 at the time of writing. You can find the available resources using

PS> Find-DscResource

Name Version ModuleName Repository

---- ------- ---------- ----------

Group 2.3.0.0 PSDscResources PSGallery

GroupSet 2.3.0.0 PSDscResources PSGallery

Registry 2.3.0.0 PSDscResources PSGallery

Script 2.3.0.0 PSDscResources PSGallery

Service 2.3.0.0 PSDscResources PSGallery

xDefaultGatewayAddress 3.1.0.0 xNetworking PSGallery

xDHCPClient 3.1.0.0 xNetworking PSGallery

xDnsClientGlobalSetting 3.1.0.0 xNetworking PSGallery

cNtfsPermissionEntry 1.3.0 cNtfsAccessControl PSGallery

cNtfsPermissionsInheritance 1.3.0 cNtfsAccessControl PSGallery

#### Note

This is a small sample of the available resources.

You’ll notice that some of the resources have a c or an x as a prefix. These prefixes indicate that a resource is supplied by the PowerShell community (c) or is a Microsoft-supplied resource that’s classed as experimental (x, meaning it may change).

Resources are delivered as modules. To use a resource from the gallery, download and install the module that contains the resource, as discussed in [chapter 9](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-9-73.xhtml#ch09). If you can’t find a ready-made resource, you can write your own—we’ll show you how to do that in [chapter 19](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-19-143.xhtml#ch19) when we discuss PowerShell classes.

#### Idempotent Operation

What happens if you run an imperative script that changes your server’s configuration and then rerun it at some future time? If you’re lucky, nothing bad happens during the rerun, but it’s quite possible for the server configuration to be damaged so that the server becomes unusable.

DSC (like all good configuration management systems) is idempotent—the configuration can be applied multiple times without changing the result beyond the initial application. For a simple example, multiplying by 1 is an idempotent operation:

PS> 9\*1

9

PS> 9\*1\*1

9

PS> 9\*1\*1\*1

9

PS> 9\*1\*1\*1\*1

9

You can multiply a number by 1 as many times as you want, and you’ll always get the same result.

Applying a DSC configuration multiple times to the same target gives the desired configuration. DSC checks whether the server is compliant with the configuration and, if so, doesn’t make any changes. We’ll show this in action in [section 18.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-182-137.xhtml#ch18lev1sec2).

#### DSC Versions

DSC was originally introduced with Windows Server 2012 R2. Since that time, a number of changes have been made to DSC that produce different versions. The major changes occurred as follows:

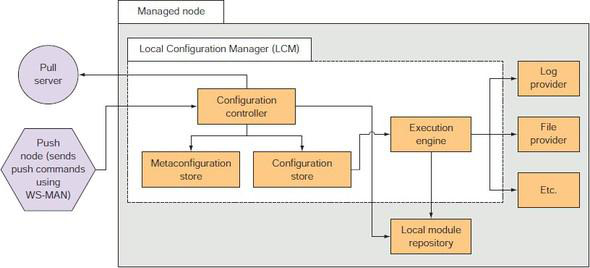
* Windows 2012 R2 RTM, July 2013
* Windows 2012 R2 General Availability, October 2013
* Windows 2012 R2 Update, November 2014
* DSC for Linux, versions 1.0 and 1.1, May 2015 and September 2015 respectively
* WMF 4.0 update for Windows 2012 and 2008 R2, January 2016
* WMF 5.0 RTM, December 2015
* Windows 2016 and WMF 5.1, October 2016

In practical terms, this means there are potential conflicts in the MOF file (usually the introduction of new properties) between these versions. The versions are backward-compatible; old versions work with new versions, but not vice versa. You need to either be consistent between the DSC versions on the machine on which you create the configuration and the machine to which you apply it, or modify the MOF as applicable to accommodate the differences. We recommend consistency between DSC versions as the safest approach.

### 18.1.3 DSC Architecture

The architecture of DSC is illustrated in [figure 18.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-181-136.xhtml#ch18fig03).

Figure 18.3: DSC architecture



The important point in the DSC architecture is the separation of the managed node (the large box in [figure 18.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-181-136.xhtml#ch18fig03)) from the source configuration (written in PowerShell) on the pull server or the push node (on the left-hand side of the figure). The server passes a static, intermediate representation of the configuration (a MOF file) to the Local Configuration Manager (LCM) on the managed node.

The LCM takes each resource in the MOF file and passes it to the corresponding resource provider. The resource providers are pieces of PowerShell code contained in modules that are responsible for ensuring that the system is compliant with the resources in the configuration.

The configuration controller does a couple of things: It validates the MOF file (valid syntax schema is correct, and so on) and it checks to see if all the necessary resource providers are available. If there are missing providers, and a pull server is configured, then the resources are downloaded. If it’s not configured to download resources, then it fails the validation process. The execution engine is only responsible for interpreting the configuration and calling the providers. If you push the configuration to the managed node, you have to ensure that the required modules are also installed on the managed node.

On Windows, the resources can be written in PowerShell or in unmanaged code as WMI providers, though that’s not encouraged—resources should be written only in PowerShell. On Linux, they are currently written in Python. Generalizing the LCM-to-resource-provider interface to allow the providers to be written in pretty much any language has been discussed by the PowerShell team, but that won’t happen in the near future.

This architecture and set of abstractions let Windows administrators manage Linux without needing any special knowledge of Linux, and vice versa. In the case of simple configurations, they can be written without even knowing much about PowerShell!

Enough theory—it’s time to see how this works in practice.

## 18.2 Push Mode to a Single Node

Push mode is the simplest way to use DSC. You can construct configurations that apply to single nodes or multiple nodes. In a single configuration file, you can set one or many configuration items with interlocking dependencies, based on the complexity of your configuration. Installing a single Windows feature may require only a single configuration item, whereas installing and configuring multiple features could require a number of configuration items.

#### Note

You’ll get most benefit from this chapter by trying these examples in your test environment. Viewing the output from the application of a configuration is useful and educational.

Push mode is the ideal place to start with DSC, but it has issues—and limitations—you need to understand. We’ll start by showing you how to create a DSC configuration for a single node.

### 18.2.1 Create Configuration

In [section 18.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-181-136.xhtml#ch18lev1sec1) we showed you an example configuration that creates a folder:

Configuration AddFile {

File TestFolder {

Ensure = 'Present'

Type = 'Directory'

DestinationPath = 'C:\TestFolder'

Force = $true

}

}

Let’s use another option: create the folder and add a file to the folder, as shown here.

#### Listing 18.1: A simple push configuration

Configuration AddFile {

Node W16TGT01 { 1

File TestFile { 2

Ensure = 'Present'

Type = 'File'

DestinationPath = 'C:\TestFolder\TestFile1.txt'

Contents = 'My first Configuration'

Force = $true

}

}

}

AddFile -OutputPath .\MOF 3

* 1 Define target name
* 2 Define file configuration
* 3 Create MOF file

The configuration is named AddFile. Configuration names are arbitrary. The Node keyword 1 defines the computer to which the configuration will be applied. If you don’t use Node, the configuration will be applied to the local machine (the MOF file will be named localhost.mof).

The configuration item 2 uses the File resource to ensure that a file named TestFile1.txt is present in the folder and has 'My first configuration' set as its contents. If the folder structure you specify in the path for your file isn’t present, the configuration will create the appropriate path.

The configuration is run 3, and a MOF file is created in the MOF subfolder of the current folder.

#### Note

If you don’t use -OutputPath when you run the configuration, your MOF file will be created in a subfolder with the same name as the configuration. How you organize your MOF files is up to you, but we recommend you decide on a method and stick with it.

### 18.2.2 MOF File Contents

Running the configuration produces the following output:

PS> AddFile -OutputPath .\MOF

WARNING: The configuration 'AddFile' is loading one or more built-

in resources without explicitly importing associated modules. Add

Import-DscResource –ModuleName 'PSDesiredStateConfiguration' to your

configuration to avoid this message.

Directory: C:\Scripts\MOF

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 05/02/2017 11:09 2868 W16TGT01.mof

You can stop the warning message appearing by adding this line of code immediately before the Node keyword:

Import-DscResource –ModuleName PSDesiredStateConfiguration

See [listing 18.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-183-138.xhtml#ch18ex04) for an example.

#### Note

You need to import any resou.rces you use other than the built-in resources, so this is a good habit to get into.

Here is the MOF file you generated.

#### Listing 18.2: MOF file created by [listing 18.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-182-137.xhtml#ch18ex01)

/\*

@TargetNode='W16TGT01'

@GeneratedBy=Richard

@GenerationDate=05/02/2017 11:09:03

@GenerationHost=W16DSC01

\*/

instance of MSFT\_FileDirectoryConfiguration as $MSFT\_FileDirectoryConfiguration1ref

{

ResourceID = "[File]TestFile";

Type = "File";

Ensure = "Present";

Contents = "My first Configuration";

DestinationPath = "C:\\TestFolder\\TestFile1.txt";

Force = True;

ModuleName = "PSDesiredStateConfiguration";

SourceInfo = "C:\\Scripts\\PIA3e\\Listing18.1.ps1::3::5::File";

ModuleVersion = "1.0";

ConfigurationName = "AddFile";

};

instance of OMI\_ConfigurationDocument

{

Version="2.0.0";

MinimumCompatibleVersion = "1.0.0";

CompatibleVersionAdditionalProperties= {"Omi\_BaseResource:ConfigurationName"};

Author="Richard";

GenerationDate="05/02/2017 11:09:03";

GenerationHost="W16DSC01";

Name="AddFile";

};

#### Note

We’ve removed some blank lines from [listing 18.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-182-137.xhtml#ch18ex02) to save space.

The MOF file starts with a header block that includes information on the target node and the machine and user that generated the file. Each resource instance has a corresponding block in the MOF file that starts

instance of MSFT\_FileDirectoryConfiguration

You can compare the information in these blocks directly with the corresponding configuration item. The CIM classes given in the MOF file can be found in the ROOT\Microsoft\Windows\DesiredStateConfiguration namespace.

The last block in the MOF file starts

instance of OMI\_ConfigurationDocument

The contents of this block vary, depending on the version of PowerShell used to generate the MOF file. If you don’t have matching DSC versions on your target machines and the machine you use to generate the MOF file, you may get an error when you apply the MOF file. The best approach is to ensure the PowerShell versions match—otherwise, you may need to edit the MOF file to remove lines that earlier versions of PowerShell can’t handle. The error message should indicate the line in the MOF file causing the problem.

The next step is to apply the MOF to the target computer.

### 18.2.3 Applying the Configuration

Once you’ve generated the MOF file, you can apply the configuration to your target machine:

PS> Start-DscConfiguration -ComputerName W16TGT01 -Path .\MOF\ `

-Wait -Verbose

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters,

''methodName' = SendConfigurationApply,'className' =

MSFT\_DSCLocalConfigurationManager,'namespaceName' = root/Microsoft/Windows/

DesiredStateConfiguration'.

VERBOSE: An LCM method call arrived from computer W16DSC01 with

user sid S-1-5-21-759617655-3516038109-1479587680-1104.

VERBOSE: [W16TGT01]: LCM: [ Start Set ]

VERBOSE: [W16TGT01]: LCM: [ Start Resource ] [[File]TestFile]

VERBOSE: [W16TGT01]: LCM: [ Start Test ] [[File]TestFile]

VERBOSE: [W16TGT01]: [[File]TestFile] The system

cannot find the path specified.

VERBOSE: [W16TGT01]: [[File]TestFile] The related

file/directory is: C:\TestFolder\TestFile1.txt.

VERBOSE: [W16TGT01]: LCM: [ End Test ] [[File]TestFile] in 0.0320

seconds.

VERBOSE: [W16TGT01]: LCM: [ Start Set ] [[File]TestFile]

VERBOSE: [W16TGT01]: [[File]TestFile] The system

cannot find the path specified.

VERBOSE: [W16TGT01]: [[File]TestFile] The related

file/directory is: C:\TestFolder\TestFile1.txt.

VERBOSE: [W16TGT01]: LCM: [ End Set ] [[File]TestFile] in 0.0000

seconds.

VERBOSE: [W16TGT01]: LCM: [ End Resource ] [[File]TestFile]

VERBOSE: [W16TGT01]: LCM: [ End Set ]

VERBOSE: [W16TGT01]: LCM: [ End Set ] in 0.3590 seconds.

VERBOSE: Operation 'Invoke CimMethod' complete.

VERBOSE: Time taken for configuration job to complete is 0.738 seconds

There are a few things to note before we discuss the output:

* The MOF file doesn’t have to be specified, just the path to it. Start-DscConfiguration figures out the correct MOF file to use based on the name of the machine you specify.
* The MOF file is transported to the target machine over WS-MAN. You can specify the target machine through the -ComputerName parameter or you can create a CIM session to the target machine. If you have a folder that contains MOF files for only the machines you want to configure, you can supply the -Path.
* You don’t need to use -Verbose all the time, but it’s a good idea when you’re testing a configuration because you can see what’s happening as the configuration is applied.
* If you don’t specify -Wait, the configuration is applied by a PowerShell job of job type ConfigurationJob. You can manage the jobs created by DSC with the standard PowerShell job cmdlets (see [chapter 13](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-13-108.xhtml#ch13)).

As you read through the output from Start-DscConfiguration, you’ll notice that you’re invoking a CIM method. The LCM on the target machine receives the configuration and tests whether the configuration matches the configuration document. In this case, the LCM is testing for the existence of the file C:\TestFolder\TestFile1.txt.

The configuration item isn’t found, so the configuration is applied—look for Start Set and End Set pairs. It’s confusing, but the application of the configuration will generate a message that the item can’t be found. It makes sense when you think that the configuration wouldn’t be applied if it already existed!

Now would be a good time to show that DSC is idempotent. Reapply the configuration:

PS> Start-DscConfiguration -ComputerName W16TGT01 -Path .\MOF\ `

-Wait -Verbose

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters,

''methodName' = SendConfigurationApply,'className' =

MSFT\_DSCLocalConfigurationManager,'namespaceName' = root/Microsoft/Windows/

DesiredStateConfiguration'.

VERBOSE: An LCM method call arrived from computer W16DSC01 with user

sid S-1-5-21-759617655-3516038109-1479587680-1104.

VERBOSE: [W16TGT01]: LCM: [ Start Set ]

VERBOSE: [W16TGT01]: LCM: [ Start Resource ] [[File]TestFile]

VERBOSE: [W16TGT01]: LCM: [ Start Test ] [[File]TestFile]

VERBOSE: [W16TGT01]: [[File]TestFile] The

destination object was found and no action is required.

VERBOSE: [W16TGT01]: LCM: [ End Test ] [[File]TestFile] in 0.0310

seconds.

VERBOSE: [W16TGT01]: LCM: [ Skip Set ] [[File]TestFile]

VERBOSE: [W16TGT01]: LCM: [ End Resource ] [[File]TestFile]

VERBOSE: [W16TGT01]: LCM: [ End Set ]

VERBOSE: [W16TGT01]: LCM: [ End Set ] in 0.3440 seconds.

VERBOSE: Operation 'Invoke CimMethod' complete.

VERBOSE: Time taken for configuration job to complete is 0.543 seconds

You’ll receive messages stating 'The destination object was found and no action is required', and you’ll see Skip Set statements

You can test whether a server is configured to match the configuration document.

### 18.2.4 Testing the Configuration Application

One test is to see if the configuration item is present:

PS> Invoke-Command -ComputerName W16TGT01 -ScriptBlock {

Get-Content -Path c:\testfolder\testfile1.txt

}

My first Configuration

Unfortunately, that doesn’t test whether the configuration is correct. The correct test is to use Test-DscConfiguration:

PS> Test-DscConfiguration -ComputerName W16TGT01

True

It would be nice to see a bit more information, so you can include the MOF file in the test:

PS> Test-DscConfiguration -ComputerName W16TGT01 `

-ReferenceConfiguration .\MOF\W16TGT01.mof |

Format-List

InDesiredState : True

ResourcesInDesiredState : {[File]TestFile}

ResourcesNotInDesiredState :

ReturnValue : 0

PSComputerName : W16TGT01

Alternatively, you can use the -Verbose parameter:

PS> Test-DscConfiguration -ComputerName W16TGT01 -Verbose

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters,

''methodName' = TestConfiguration,'className' =

MSFT\_DSCLocalConfigurationManager,'namespaceName' = root/Microsoft/Windows/

DesiredStateConfiguration'.

VERBOSE: An LCM method call arrived from computer W16DSC01 with user sid S-1-

5-21-759617655-3516038109-1479587680-1104.

VERBOSE: [W16TGT01]: LCM: [ Start Test ]

VERBOSE: [W16TGT01]: LCM: [ Start Resource ] [[File]TestFile]

VERBOSE: [W16TGT01]: LCM: [ Start Test ] [[File]TestFile]

VERBOSE: [W16TGT01]: [[File]TestFile] The

destination object was found and no action is required.

VERBOSE: [W16TGT01]: LCM: [ End Test ] [[File]TestFile] True in

0.0310 seconds.

VERBOSE: [W16TGT01]: LCM: [ End Resource ] [[File]TestFile]

VERBOSE: [W16TGT01]: LCM: [ End Test ] Completed processing test

operation. The operation returned True.

VERBOSE: [W16TGT01]: LCM: [ End Test ] in 0.0630 seconds.

VERBOSE: Operation 'Invoke CimMethod' complete.

True

VERBOSE: Time taken for configuration job to complete is 0.201 seconds

The output shows that the configuration items were found and the configuration is correct.

### 18.2.5 Viewing the Current Configuration

You can view the current configuration of the target machine. In this case we’ll use a CIM session:

PS> $cs = New-CimSession -ComputerName W16TGT01

PS> Get-DscConfiguration -CimSession $cs

#### CimSession and ComputerName

You may want to use a CIM session if you’re going to be performing multiple actions against the target machines—for instance, setting, testing, and getting the configuration.

You can simplify the approach because the -CimSession parameter on Get-Dsc-Configuration (and other cmdlets that have a -CimSession parameter) will take an array of computer names (or a single computer name) instead of a CIM session object. If you use a computer name, a CIM session will be created, used, and destroyed in the background. This is approach is fine if you’re performing a single action, but a CIM session is recommended if you’re performing multiple actions because it’s a more efficient technique.

For each configuration item in the configuration document, you’ll see output of this form:

ConfigurationName : AddFile

DependsOn :

ModuleName : PSDesiredStateConfiguration

ModuleVersion :

PsDscRunAsCredential :

ResourceId : [File]TestFile

SourceInfo :

Attributes : {archive}

Checksum :

Contents :

CreatedDate : 02/05/2017 11:09:47

Credential :

DestinationPath : C:\TestFolder\TestFile1.txt

Ensure : present

Force :

MatchSource :

ModifiedDate : 02/05/2017 11:09:47

Recurse :

Size : 25

SourcePath :

SubItems :

Type : file

PSComputerName : W16TGT01

CimClassName : MSFT\_FileDirectoryConfiguration

Don’t forget to remove the CIM session if you don’t need it:

PS> Remove-CimSession -CimSession $cs

You’ve seen how to apply and test a configuration. The last part of the lifecycle is to remove the configuration.

### 18.2.6 Removing a Configuration

There will come a time when you need to remove the configuration items from your target because you’re repurposing the machine or the configuration is no longer appropriate. In the case of a file, you could perform a deletion, but it’s better practice to reverse the configuration. This shows the reversal of the configuration from [listing 18.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-182-137.xhtml#ch18ex01).

#### Listing 18.3: Removing a configuration

Configuration AddFile {

Node W16TGT01 {

File TestFile {

Ensure = 'Absent' 1

Type = 'File'

DestinationPath = 'C:\TestFolder\TestFile1.txt'

Force = $true

}

File TestFolder {

Ensure = 'Absent' 1

Type = 'Directory'

DestinationPath = 'C:\TestFolder'

Force = $true

DependsOn = '[File]TestFile' 2

}

}

AddFile -OutputPath .\MOF

* 1 Ensure item removal
* 2 Remove file before folder

Two important points to note. First, the Ensure parameter is set to Absent 1. This ensures that the item is removed if present. Second, the folder removal should be dependent 2 on the file removal. Once the MOF file is created you can apply it:

PS> Start-DscConfiguration -ComputerName W16TGT01 -Path .\MOF\ -Wait

Use Test-DscConfiguration to determine if the file has been removed. As a second check, you can use Test-Path:

PS> Test-Path -Path \\W16TGT01\C$\TestFolder\TestFile1.txt

False

We’ve spent quite a long time walking you through creating, applying, testing, and deleting a configuration. This only applied to a single machine. It’s more likely that you’ll want to apply a configuration to multiple machines—preferably simultaneously.

## 18.3 Pushing to Multiple Nodes

If you need to apply the same configuration to multiple machines, you could run [listing 18.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-182-137.xhtml#ch18ex01) a number of times, changing the computer name each time. That’s inefficient, not to mention boring and error-prone, so we’ll show you how to parameterize your configurations. First, we’ll show how to change only the nodes to which you’ll apply the configuration. Then we’ll show you how to use configuration metadata to change the configuration being applied based on the machine name.

### 18.3.1 Parameterizing the Computer Name

If you have a number of machines you need to apply exactly the same configuration to, the easiest approach is to parameterize the computer name.

#### Listing 18.4: Parameterizing the computer name

Configuration AddFile {

param (

[Parameter(Mandatory=$true)]

[string[]]$ComputerName 1

)

Import-DscResource –ModuleName PSDesiredStateConfiguration

Node $ComputerName { 2

File TestFile {

Ensure = 'Present'

Type = 'File'

DestinationPath = 'C:\TestFolder\TestFile1.txt'

Contents = 'My first Configuration'

Force = $true

}

}

}

AddFile -OutputPath .\MOF -ComputerName 'W16TGT01', 'W16DSC02' 3

* 1 Parameter block
* 2 Node uses parameter
* 3 Computer names supplied

The configuration is an evolution of [listing 18.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-182-137.xhtml#ch18ex01). The parameter block 1 defines a single mandatory parameter—ComputerName—which is an array of strings, each element of which is a computer name. A statement to explicitly import the resources being used has been added:

Import-DscResource –ModuleName PSDesiredStateConfiguration

This will stop the warning messages being issued that we saw with [listing 18.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-182-137.xhtml#ch18ex01). The Node 2 is modified to use the ComputerName parameter name rather than having a hardcoded computer name.

When the configuration is run 3, the -ComputerName parameter is used to supply the names of the computers you’ll apply the configuration to. You’ll see output similar to this:

Directory: C:\scripts\MOF

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 02/05/2017 14:50 2128 W16TGT01.mof

-a---- 02/05/2017 14:50 2128 W16DSC02.mof

A MOF file is produced for each computer name that you supply to the configuration. Notice that you didn’t have to create any looping structures in your code to manage multiple machines—it’s all done for you.

#### Note

If you think the parameter block looks like that used in functions and scripts, you’re correct.

You can now apply your configuration:

PS> Start-DscConfiguration -ComputerName W16TGT01, W16DSC02 `

-Path .\MOF\ -Wait

If you don’t use the -Wait parameter, you’ll only see a single job managing the application of the configuration. But if you look at the child jobs

PS> Get-Job -IncludeChildJob

Id Name PSJobTypeName State HasMoreData Location Command

-- ---- ------------- ----- ----------- -------- -------

22 Job22 ConfigurationJob Completed True W16TGT01,W16DSC02 Sta...

23 Job23 ConfigurationJob Completed True W16TGT01 Sta...

24 Job24 ConfigurationJob Completed True W16DSC02 Sta...

you’ll see that there is one child job per machine to be configured. The parent job manages the creation and running of the child jobs.

Test the application of the configuration:

PS> Test-DscConfiguration -ComputerName W16TGT01, W16DSC02

True

True

If you need more details on the applied configurations:

PS> Get-DscConfiguration -CimSession W16TGT01, W16DSC02 |

Format-Table PSComputerName, ConfigurationName, Ensure, Type -AutoSize

PSComputerName ConfigurationName Ensure Type

-------------- ----------------- ------ ----

W16TGT01 AddFile present file

W16DSC02 AddFile present file

We’ll leave the creation of the configuration to remove the folders and file to you (hint: modify [listing 18.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-182-137.xhtml#ch18ex03)). A version of the code is available in the book’s download file: RemoveListing18.4.ps1.

As well as parameterizing the computer names, you can also supply other information to the configuration, including the parts of the configuration to apply.

### 18.3.2 Using Configuration Data

You can parameterize your configurations beyond the computer name by supplying configuration data in the form of hashtables. In fact, you can configure anything you want using parameters. But the purpose of configuration data is to allow you to easily separate configuration (also known as environment configuration) from topology (structural configuration). You can also think of it as separating the what (topology) from the where (configuration). This is an important point that people seem to miss. The canonical example is <http://mng.bz/3LsX>. It defines three roles (configurations) and then uses configuration data to map those roles to physical machines. All three roles can be on one machine, or each role can be on a discrete machine or on multiple machines for High Availability scenarios.

This example modifies the contents of the text file depending on the machine being configured.

#### Listing 18.5: Using configuration metadata

$ConfigurationData = @{ 1

AllNodes = @(

@{NodeName = 'W16TGT01';FileText='Configuration for Role 1'},

@{NodeName = 'W16DSC02';FileText='Configuration for Role 2'}

)

}

Configuration AddFile {

Import-DscResource –ModuleName PSDesiredStateConfiguration

Node $AllNodes.NodeName { 2

File TestFile {

Ensure = 'Present'

Type = 'File'

DestinationPath = 'C:\TestFolder\TestFile1.txt'

Contents = $Node.FileText 3

Force = $true

}

}

}

AddFile -OutputPath .\MOF -ConfigurationData $ConfigurationData 4

* 1 Configuration data
* 2 Setting node name
* 3 Setting text
* 4 Running configuration

Configuration data 1 is supplied as a hashtable that must have one key named AllNodes. Other keys are permitted, though seldom used. AllNodes is an array of hashtables; each hashtable defines the configuration for a single machine and must have a key named NodeName (name of the machine to be configured). Again, other keys are permitted.

You can define the configuration data hashtable in a .psd1 file and access it as

AddFile -OutputPath .\MOF -ConfigurationData ./confdata.psd1

#### Note

Forgetting to include the -ConfigurationData parameter and the hashtable is a common error when you start using this approach.

When the configuration is run, $AllNodes.NodeName 2 is accessed to process each individual machine affected by the configuration. As the configuration is processed, other elements of the machine’s hashtable, as defined in the configuration data, are accessed—for instance, $Node.FileText 3 to set the file’s contents.

The configuration data is linked to the configuration when it’s run using the -ConfigurationData parameter 4 and passing the variable containing the configuration hashtable. A MOF file is produced for each machine listed in the configuration data, as you would expect.

Applying the configuration is performed in the usual way—supply the computer names or a CIM session together with the path to the MOF files:

PS> Start-DscConfiguration -ComputerName W16TGT01, W16DSC02 `

-Path .\MOF\ -Wait -Verbose

You can test the configurations individually:

PS> Test-DscConfiguration -ComputerName W16TGT01

True

PS> Test-DscConfiguration -ComputerName W16DSC02

True

or simultaneously:

PS> Test-DscConfiguration -ComputerName W16TGT01, W16DSC02

True

True

The final test is to view the content of the files:

PS> Invoke-Command -ComputerName W16TGT01, W16DSC02 `

-ScriptBlock {Get-Content -Path C:\TestFolder\TestFile1.txt}

Configuration for Role 1

Configuration for Role 2

You can remove this configuration, if required, using the RemoveListing18.4.ps1 script in the download code.

So far, you’ve seen how to apply a configuration to multiple machines. There are many situations where you need to create a set of machines, each of which has its own unique requirements.

### 18.3.3 Configuration Data and Roles

Imagine that you’re creating the infrastructure for an internet-facing system such as an e-commerce site. You’d need to create a number of identical web servers, a server to run your business logic, and possibly a database server. Also, you’ll need to rebuild this infrastructure on a frequent basis as new versions of the software are released.

You could set up a single configuration for each server type and run that. In fact, that’s how you would probably start for development purposes. But you should look at a single configuration that works with all your servers and applies the correct configuration based on the role of the server.

This is easier to grasp with an example. This listing shows a role-based configuration.

#### Listing 18.6: Role-based configurations

$ConfigurationData = @{ 1

AllNodes = @(

@{NodeName = 'W16TGT01';Role = 'Hyper-V'},

@{NodeName = 'W16CN01';Role = 'AD'}

)

}

Configuration RoleConfiguration 2

{

param ($Role)

switch ($Role) {

'Hyper-V' {

Import-DscResource -ModuleName PSDesiredStateConfiguration

WindowsFeature Hyper-V {

Ensure = 'Present'

Name = 'Hyper-V-PowerShell'

}

}

'AD' {

Import-DscResource -ModuleName PSDesiredStateConfiguration

WindowsFeature AD {

Ensure = 'Present'

Name = 'RSAT-AD-PowerShell'

}

}

}

}

Configuration ToolsConfig 3

{

Import-DscResource -ModuleName PSDesiredStateConfiguration

node $allnodes.NodeName

{

RoleConfiguration ServerRole

{

Role = $Node.Role

}

}

}

ToolsConfig -ConfigurationData $ConfigurationData `

-OutputPath .\MOF 4

* 1 Configuration metadata
* 2 Composite resource
* 3 Main configuration
* 4 Run configuration

We start with the configuration metadata 1 held within the $ConfigurationData hashtable. The metadata defines the server name and the role it will take. The role controls the configuration applied to the server.

A configuration called RoleConfiguration 2 performs the configuration. It takes a role as a parameter, and using a switch parameter determines which Windows feature—Hyper-V PowerShell module or Active Directory PowerShell module—is installed. You could have further nesting at this point by calling additional configurations. As with all nesting options, achieving the correct balance between granularity and maintainability will depend on your exact circumstances and the scenarios you’re working with.

#### Composite resources

A composite resource is a DSC configuration that’s used as a resource in another configuration. In the case of [listing 18.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-183-138.xhtml#ch18ex06), the configuration RoleConfiguration performs the task of configuring the target based on the role assigned to that system. RoleConfiguration is used as a composite resource by the ToolsConfig configuration.

In this case, the composite configuration is contained in the same file as the top-level configuration. If you wanted to reuse the composite resource in many other different configurations, you could save it with a .schema.psm1 extension. You’d also need to create a module manifest that defined the .schema.psm1 file as the root module.

A worked example of using composite resources in this manner is available at <http://mng.bz/1e6G>.

The ToolsConfig configuration 3 is the master configuration that calls RoleConfiguration. ToolsConfig is the configuration that’s run 4 and to which the configuration data is passed.

Once the MOF files are created, the configuration can be applied:

PS> Start-DscConfiguration -ComputerName W16TGT01, W16CN01 `

-Path .\MOF\ -Wait -Verbose

If you watch the output, you’ll see these two lines:

VERBOSE: [W16CN01]: [[WindowsFeature]AD::[RoleConfiguration]ServerRole]

Successfully installed the feature RSAT-AD-PowerShell.

VERBOSE: [W16TGT01]:

[[WindowsFeature]Hyper-V::[RoleConfiguration]ServerRole]

Successfully installed the feature Hyper-V-PowerShell.

They indicate that the configuration has been successfully applied.

Testing the configuration is a little more difficult:

PS> Invoke-Command -ComputerName W16TGT01, W16CN01 -ScriptBlock {

Get-WindowsFeature -Name Hyper-V-PowerShell, RSAT-AD-PowerShell

} | sort Name |

Format-Table Name, DisplayName, Installed, PSComputerName

Name DisplayName Installed PSComputerName

---- ----------- --------- --------------

Hyper-V-PowerShell Hyper-V Module... True W16TGT01

Hyper-V-PowerShell Hyper-V Module... False W16CN01

RSAT-AD-PowerShell Active Directo... True W16CN01

RSAT-AD-PowerShell Active Directo... False W16TGT01

You can see from the output that the correct configuration has been applied to each machine.

Even with parameterization, push mode has a number of issues that limit its usefulness.

### 18.3.4 Issues with Push Mode

You’ve been introduced to DSC in push mode and the benefits you gain in terms of managing your server configurations. DSC push mode is a huge step forward compared to manually configuring servers, but as with most things, it has its minuses.

Here are the main drawbacks to using push mode:

* **Doesn’t scale—** Using push mode on 10 servers is manageable. At a scale of hundreds or thousands of servers, manual processes break down. A situation with frequent builds required by new application versions also causes push mode to be unsatisfactory.
* **Delivering resource modules to target—** The PowerShell module containing the DSC resources used by a configuration has to be installed on the target machine. When using push mode, it’s your responsibility to ensure this (hint: copying files over a PowerShell remote session is a great way to perform this action).
* **Fire and forget—** Monitoring and reporting are manual processes. Once you’ve pushed the configuration to the target node, that’s it. All finished. If you want to monitor the configuration and correct any configuration drift, it’s your job to figure out how to do that and create the required scripts.

These points bring us to the conclusion that DSC push mode is great for development and testing. It’s also adequate for small environments. But if you have a large environment to manage through DSC, or you have frequent software releases for which you need to build new infrastructure each time, you need something more. That something is DSC pull mode.

## 18.4 DSC in Pull Mode

In the DSC examples you’ve seen so far, the configuration has been pushed to the target server. As you saw in [section 18.3.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-183-138.xhtml#ch18lev2sec13), push mode doesn’t scale well. In this section, we’ll cover DSC in pull mode, where the target server contacts the pull server and pulls—and then applies its configuration.

We’ll start by covering the pull server architecture and then move on to showing you how to create a pull server using DSC. When the pull server is running, you need to create your configuration and publish the MOF file (together with any required modules) to the pull server.

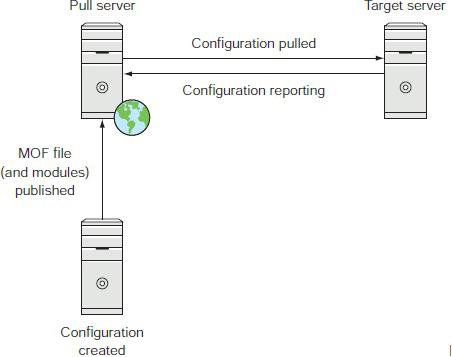
The final part of the picture is to configure the target machine’s LCM to work with the pull server, which we’ll postpone to [section 18.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-185-140.xhtml#ch18lev1sec5).

What does DSC look like in pull mode?

### 18.4.1 Pull Server Architecture

The architecture of DSC in pull mode is illustrated in [figure 18.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-184-139.xhtml#ch18fig04).

Figure 18.4: DSC in pull mode



A DSC configuration is created. This is usually on a separate machine from the pull server—development on a production server is a bad thing in many organizations. The configuration is run to create a MOF file. The MOF file, together with any required DSC resource modules, is published to the pull server.

#### Note

The pull server protocol specification is available through the Microsoft Open Specifications program. Its designation is MS-DSCPM. See <http://mng.bz/TzY3> for the specification of the protocol.

The target server is configured via the LCM to periodically poll the pull server for its configurations. When a configuration is found, it and any associated resource modules are downloaded to the target server and applied.

#### Note

LCM configuration is covered in [section 18.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-185-140.xhtml#ch18lev1sec5).

The target server can be configured to report the status of its configuration to the pull server. The LCM can also be configured to reapply the configuration if the target server’s configuration drifts from the desired state (you can also do this in push mode, but it’s rare to see that done). These mechanisms provide a compliance regime for your environment. You can confidently state what a server’s configuration should be and prove that its state matches the desired state. PowerShell 1, Auditors 0.

The next big question is: How do you create a pull server?

### 18.4.2 Creating a Pull Server

The best way to create a DSC pull server is to use DSC. In this section, we’ll show you how to create a web-based pull server.

#### Pull server on SMB Share

You can set up a DSC pull server based on an SMB share instead: <http://mng.bz/uWRh>.

Our simple advice is don’t.

An SMB share–based pull server isn’t as versatile as the full pull server we’re going to show you and should only be used for testing the pull concept or for situations where it’s impossible to use an HTTP-based pull server.

If you have a machine you’re using as a push server, then you can create the configuration and push it to the relevant server. If you have a pull server already in your environment and want to create another one, get the new pull server to pull its configuration from the original pull server.

Before you can create the pull server, you need to take care of a few prerequisites.

#### Pull Server Prerequisites

There are two main prerequisites. First, if you want to secure and encrypt the web traffic to and from your pull server, you need to install an SSL certificate on the machine. This will be used during the creation of the pull server. You’ll need to know the thumbprint of the certificate:

PS> Get-ChildItem -Path Cert:\LocalMachine\My\

PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\My

Thumbprint Subject

---------- -------

FF24E1BA4B32D2F75A8F9648DECC1D070F1F2B13 CN=W16DSC02

Second, you’ll need to install the modules containing the DSC resources you require to install and configure the pull server. These modules can be found on the PowerShell Gallery.

The following modules are required:

PS C:\Scripts> Find-Module xPSDesiredStateConfiguration

Version Name Repository Description

------- ---- ---------- -----------

5.1.0.0 xPSDesiredStateConfiguration PSGallery The xPSD...

PS> Find-Module xWebAdministration

Version Name Repository Description

------- ---- ---------- -----------

1.16.0.0 xWebAdministration PSGallery Module w...

Both xPSDesiredStateConfiguration and xWebAdministration are classed as experimental and as such are subject to change, including breaking changes, with no notice. If the versions you find are different from those mentioned, you’ll need to test the code we use to ensure there haven’t been any breaking changes.

These two modules need to be installed on the pull server and any system you use for creating configurations:

PS> Install-Module -Name xPSDesiredStateConfiguration, xWebAdministration ` -Force

The modules will be installed to C:\Program Files\WindowsPowerShell\Modules, as are all modules obtained from the PowerShell Gallery.

Now it’s time to create the configuration for your pull server.

#### Pull server and Local Configuration Manager

You need to do three things to configure your DSC environment to use a pull server:

1. You need to configure a pull server—which we’ll be covering in a moment.
2. You’ll need to create a configuration to operate in pull mode and publish it to the pull server. That will be covered in [section 18.4.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-184-139.xhtml#ch18lev2sec16)
3. You’ll need to configure the LCM on the target machine(s) to use the pull server. We’ll postpone that discussion until [section 18.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-185-140.xhtml#ch18lev1sec5), where we cover all aspects of the LCM.

#### Creating the Pull Server

Using a DSC configuration to create a DSC pull server is a fitting way to proceed.

#### Listing 18.7: Creating a pull server

$ConfigurationData=@{ 1

AllNodes = @(

@{

NodeName = 'W16DSC02'

Role = @('Web', 'PullServer')

CertThumbPrint = Invoke-Command -Computername 'W16DSC02' -ScriptBlock {

Get-Childitem -Path Cert:\LocalMachine\My |

where Subject -Like 'CN=W16DSC02\*' |

Select-Object -ExpandProperty ThumbPrint}

}

);

}

Configuration Pullserver {

Import-DscResource -ModuleName PSDesiredStateConfiguration 2

Import-DscResource -ModuleName xPSDesiredStateConfiguration

Import-DscResource -ModuleName xWebAdministration

Node $AllNodes.where{$\_.Role -eq 'Web'}.NodeName { 3

WindowsFeature IIS {

Ensure = "Present"

Name = "Web-Server"

}

WindowsFeature NetExtens4 { 4

Ensure = "Present"

Name = "Web-Net-Ext45"

DependsOn = '[WindowsFeature]IIS'

}

WindowsFeature AspNet45 {

Ensure = "Present"

Name = "Web-Asp-Net45"

DependsOn = '[WindowsFeature]IIS'

}

WindowsFeature ISAPIExt {

Ensure = "Present"

Name = "Web-ISAPI-Ext"

DependsOn = '[WindowsFeature]IIS'

}

WindowsFeature ISAPIFilter {

Ensure = "Present"

Name = "Web-ISAPI-filter"

DependsOn = '[WindowsFeature]IIS'

}

WindowsFeature DirectoryBrowsing { 5

Ensure = "Absent"

Name = "Web-Dir-Browsing"

DependsOn = '[WindowsFeature]IIS'

}

WindowsFeature StaticCompression {

Ensure = "Absent"

Name = "Web-Stat-Compression"

DependsOn = '[WindowsFeature]IIS'

}

WindowsFeature Management { 6

Name = 'Web-Mgmt-Service'

Ensure = 'Present'

DependsOn = @('[WindowsFeature]IIS')

}

Registry RemoteManagement {

Key = 'HKLM:\SOFTWARE\Microsoft\WebManagement\Server' 7

ValueName = 'EnableRemoteManagement'

ValueType = 'Dword'

ValueData = '1'

DependsOn = @('[WindowsFeature]IIS','[WindowsFeature]Management')

}

Service StartWMSVC {

Name = 'WMSVC'

StartupType = 'Automatic'

State = 'Running'

DependsOn = '[Registry]RemoteManagement'

}

xWebsite DefaultSite {

Name = "Default Web Site"

State = "Started"

PhysicalPath = "C:\inetpub\wwwroot"

DependsOn = "[WindowsFeature]IIS"

}

}

Node $AllNodes.where{$\_.Role -eq 'PullServer'}.NodeName { 8

WindowsFeature DSCServiceFeature {

Ensure = "Present"

Name = "DSC-Service"

}

xDscWebService DSCPullServer {

Ensure = "Present"

EndpointName = "PullServer"

Port = 8080

PhysicalPath = "$env:SystemDrive\inetpub\wwwroot\PullServer"

CertificateThumbPrint = $Node.CertThumbprint

ModulePath = "$env:PROGRAMFILES\WindowsPowerShell\DscService\Modules"

ConfigurationPath = "$env:PROGRAMFILES\WindowsPowerShell\DscService\Configuration"

State = "Started"

UseSecurityBestPractices = $false

DependsOn = "[WindowsFeature]DSCServiceFeature"

}

xDscWebService DSCComplianceServer {

Ensure = "Present"

EndpointName = "ComplianceServer"

Port = 9080

PhysicalPath = "$env:SystemDrive\inetpub\wwwroot\ComplianceServer"

CertificateThumbPrint = "AllowUnencryptedTraffic"

State = "Started"

UseSecurityBestPractices = $false

DependsOn = ("[WindowsFeature]DSCServiceFeature","[xDSCWebService]DSCPullServer")

}

}

}

Pullserver -ConfigurationData $ConfigurationData -outputPath .\MOF

* 1 Configuration data
* 2 Required resource modules
* 3 Install IIS
* 4 IIS sub-features to install
* 5 IIS sub-features to block
* 6 IIS management
* 7 Registry configuration
* 8 Install DSC

This long configuration breaks down into a number of chunks. The first chunk 1 is the configuration data. In this example, we’re setting two roles—Web and PullServer—for our server, called W16DSC02 in this case. The certificate thumbprint for the SSL certificate on the pull server is recovered through a script rather than hardcoding. This makes your code more portable and saves the error-prone exercise of typing in the thumbprint. Let PowerShell do the work for you.

Moving on to the configuration itself, the first step 2 is to import the modules required by the configuration.

#### Note

You should have installed these modules in the previous section. If you haven’t, make sure they’re installed on both the authoring server and the machine that will become your pull server.

Creating a pull server requires IIS and DSCServiceFeature to be installed. We’ll start with IIS 3 and install the basic web server Windows feature. Unfortunately, this won’t give us quite what we need, so we have to ensure that a number of the IIS sub-features are present. 4 These sub-features include ISAPI and ASP. Likewise, there are a number of IIS features we don’t want installed—ever. These include 5 Directory Browsing and Static Compression.

#### Note

We’ve shown only a few features that we don’t want installing. Your organization may have others that it thinks shouldn’t be installed. The pattern shown in [listing 18.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-184-139.xhtml#ch18ex07) is infinitely adaptable. You can add more configuration items as required.

It’s always a good idea to be able to manage your servers remotely, so make sure that you install the management service 6 and set the registry keys 7 to enable this scenario. The IIS configuration concludes by setting the IIS service (WMSVC) startup to automatic and ensures that the default website—which DSC uses—will be started when IIS starts.

The DSC configuration 8 is simpler. The first part ensures that the DSC service is installed. The pull server—DSCPullServer—is then configured. Most of the settings should be self-explanatory by now. The port that the pull server clients will use to connect is set to 8080. This is arbitrary; you can use another port if required. A specific port is specified to separate traffic if another application is using the default website. The ModulePath and ConfigurationPath settings control where the pull server stores configurations and modules that its clients need to pull. If you don’t put them in that place, the client won’t find them.

It may seem odd to set UseSecurityBestpractices to $false. Setting this property to $true will reset registry values, which in this case will be controlling SSL, under "HKLM:\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL". This environment change enforces the use of a stronger encryption cypher and may affect legacy applications. More information can be found at <http://mng.bz/U3dr> and <http://mng.bz/747N>.

#### Note

You’ll have noticed the heavy use of DependsOn in [listing 18.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-184-139.xhtml#ch18ex07). It’s worth tracing the dependencies and the interaction of the IIS and DSC configurations with each other and the configuration data. This is a complicated configuration, and if you can understand this, you’re well on the way to mastering DSC.

The final part of the DSC configuration is for the compliance server. This provides reporting and compliance information on the configuration of the pull server’s client machines. For now, we’re only configuring the compliance server. The configuration of the compliance server is similar to the pull server, but we’re allowing unencrypted traffic to use the compliance server and we’re configuring a different port.

The last line of [listing 18.7](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-184-139.xhtml#ch18ex07) runs the configuration and creates the MOF file as usual. You then need to push the configuration to the machine you’re creating a pull server on:

PS> Start-DscConfiguration -ComputerName W16DSC02 -Path .\MOF\ `

-Wait -Verbose

Expect it to run a while. Restart the new pull server:

PS> Restart-computer -ComputerName W16DSC02 -Wait -Force

Your DSC pull server should now be ready for use. Before jumping into using the pull server, we should test the configuration:

PS> Test-DscConfiguration -ComputerName W16DSC02

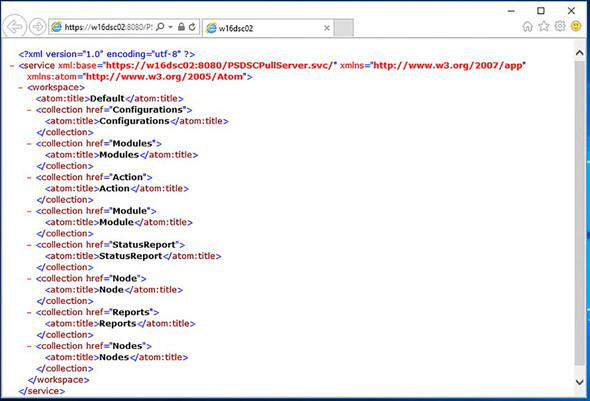
You should also test the pull server by connecting to the web service. From your authoring server (or another machine in the domain):

PS> Start-Process `

-FilePath iexplore.exe https://W16DSC02:8080/PSDSCPullServer.svc

You should see something like [figure 18.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-184-139.xhtml#ch18fig05).

Figure 18.5: Testing the pull server



Notice the name of the service: PSDSCPullServer.svc. It’s hardcoded into the DSC resource. If you need to modify that, check carefully that you find all the places it’s specified.

Your pull server is up and running. It appears to be working correctly. The next step is creating and publishing a MOF file to the pull server.

### 18.4.3 Publishing a MOF File

When you’re working with DSC in push mode, you create a MOF file and manually push it to the target server. You’re implicitly assuming that the server is running and ready to receive its configuration. When working in pull mode you don’t care about the state of the target server because you publish the MOF file to the pull server and leave the target server to pull its configuration when it’s ready.

A number of steps are required to create a configuration to pull:

1. Create the configuration.
2. Copy to the pull server.
3. Ensure resources are on the pull server.
4. Force pull (for demo or testing). In production, wait for the target machine to be ready to pull its configuration.

Examples are everything, so we’ll demonstrate this process by creating a MOF file that’ll configure the target server with a file share. The configuration is shown in [listing 18.8](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-184-139.xhtml#ch18ex08).

Before you start to create the configuration file, you need to ensure that required resources are on the authoring server. In this case, you’ll need to download the xSmbShare resource module from the PowerShell gallery:

PS> Install-Module -Name xSmbShare -Force

Once that’s installed, you have everything you need to create the configuration to be pulled.

#### Listing 18.8: Configuration to be pulled

Configuration stdShare {

param

(

[Parameter(Mandatory=$true)]

[string[]]$ComputerName

)

Import-DscResource -ModuleName PSDesiredStateConfiguration 1

Import-DscResource -ModuleName xSmbShare

Node $ComputerName {

File TestFolder { 2

Ensure = 'Present'

Type = 'Directory'

DestinationPath = 'C:\TestFolder'

Force = $true

}

File TestFile { 3

Ensure = 'Present'

Type = 'File'

DestinationPath = 'C:\TestFolder\TestFile1.txt'

Contents = 'My first Configuration'

Force = $true

}

xSmbShare StandardShare 4

{

Ensure = "Present"

Name = "Standard"

Path = 'C:\TestFolder'

Description = "This is a test SMB Share"

ConcurrentUserLimit = 0

}

}

}

stdShare -ComputerName W16TGT01 -OutputPath .\MOF

* 1 Import resources
* 2 Create folder
* 3 Create file
* 4 Create share

You’ve already seen this configuration earlier in the chapter. Creating a file in a folder was our first simple configuration example. The configuration starts by importing the required resources 1. Creating the folder 2 and its associated file 3 are performed in two steps this time. Creating the share is equally straightforward 4. A name for the share and the path to the folder are required. The description and concurrent user limits are optional.

Run the configuration to create your MOF file. If you were using push mode, you’d use Start-DscConfiguration to push the MOF file to the target machine (assuming you’d remembered to copy the required resource module to the target). With push mode, you have a few more hoops to jump through.

You need to rename the MOF file so that it matches the identification of the target server. A GUID is used as the identifier. You generate a new GUID by using New-Guid:

PS> $psclientid = New-Guid | select -ExpandProperty guid

PS> $psclientid

5827c542-20bb-487c-89cb-484cbe5f0b1f

#### Getting the GUID in production

The important thing is that the same GUID is used for the target machine’s configuration ID and the data sent to the pull server. If your target machine has already been configured to use the pull server, it’ll have a configuration ID, so you can find the GUID like this:

PS> $pscs = New-CimSession -ComputerName <target machine>

PS> $psclientid = Get-DscLocalConfigurationManager -CimSession $pscs |

select -ExpandProperty ConfigurationID

We’ll use this GUID to rename the MOF file:

PS> Get-ChildItem -Path C:\scripts\MOF\W16TGT01.mof |

Rename-Item -NewName "C:\scripts\MOF\$psclientid.mof"

You then need to create a checksum of the MOF file:

PS> New-DscChecksum -Path "C:\scripts\MOF\$psclientid.mof" -Force

The MOF and checksum files

5827c542-20bb-487c-89cb-484cbe5f0b1f.mof

5827c542-20bb-487c-89cb-484cbe5f0b1f.mof.checksum

need to be transferred to the pull server.

#### Note

The checksum files are used by the target machine when it pulls its configuration, and any required resource modules, to ensure that the integrity of those items hasn’t been compromised.

The ability to copy files across a PowerShell remoting session makes life easier here:

PS> $s = New-PSSession -ComputerName W16DSC02

PS> Get-ChildItem -Path .\MOF\ -Filter "$psclientid.\*" |

Copy-Item -Destination "C:\program Files\WindowsPowerShell\DscService\Configuration" `

-ToSession $s -Force

The last step in preparing the configuration is to get a copy of the resource module onto the pull server. This has to be archived into a zip file with a name that includes the module version. It’s always advisable to use the module you’ve installed on your authoring server to ensure there are no compatibility issues. First, get the module path and version:

PS> $module = Get-Module -ListAvailable xSmbShare

PS> $modulepath = "$(Split-Path -Path $module.Path)\\*"

PS> $moduleversion = $module.Version.ToString()

Then create a zip file and generate a checksum:

PS> Compress-Archive -Path $modulepath `

-DestinationPath "C:\scripts\ModuleZips\xSMBShare\_$moduleversion.zip" `

-Force

PS> New-DscChecksum `

-Path "C:\scripts\ModuleZips\xSMBShare\_$moduleversion.zip" `

-Force

The destination isn’t important. Keeping the zip files together helps organize the files.

#### Note

The archive cmdlets were introduced in PowerShell v5. The archive module is now part of the PowerShell open source projects on GitHub.

Your final step is to copy the module’s zip and checksum files to the pull server:

PS> Get-ChildItem `

-Path "C:\scripts\ModuleZips\xSMBShare\_$moduleversion.\*" |

Copy-Item `

-Destination "C:\Program Files\WindowsPowerShell\DscService\Modules\" `

-ToSession $s -Force

The pull server has now been configured and has a configuration ready to be pulled. It’s now time to configure the target machine to use the pull server, but before moving on to that, don’t forget to remove any unwanted PowerShell remoting sessions you created in this section.

## 18.5 Configuring the Local Configuration Manager

The previous sections in this chapter have been concerned with supplying the configuration you’ll apply to your target server, either through push or pull mode. In this section, we’ll concentrate on the target machine and, more specifically, on the LCM.

#### Note

This section is specifically targeted at PowerShell v5. For LCM configuration in PowerShell v4, see <http://mng.bz/A84O>.

Every target machine has an independent LCM. It is DSC’s local engine with responsibility for applying the configurations received by the machine. It also controls the following:

* Setting how the machine receives configurations—push or pull mode
* Timing with which the machine pulls and applies configurations
* Controlling pull servers used by the machine
* Controlling reporting servers used by the machine
* Managing partial configurations (see [section 18.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-186-141.xhtml#ch18lev1sec6))

We’ll look at the default LCM settings and see what can be changed and how those changes can be applied. Then we’ll show you how to use DSC to configure the LCM to use a pull server.

### 18.5.1 LCM Settings

All machines running Windows PowerShell v4 and above have a copy of the LCM running. It’s part of the operating system—or it’s installed through the Windows Management Framework if you’ve upgraded PowerShell on an older copy of Windows. We’ll show you the default LCM settings in this section and then see how to change them.

#### Default LCM Settings

The default LCM settings can be viewed on a newly created machine (or one you can guarantee hasn’t been modified):

PS> $cs = New-CimSession -ComputerName W16TGT01

PS> Get-DscLocalConfigurationManager -CimSession $cs

ActionAfterReboot : ContinueConfiguration

AgentId : CBE8E714-C86B-11E6-841F-00155D36C90B

AllowModuleOverWrite : False

CertificateID :

ConfigurationDownloadManagers : {}

ConfigurationID :

ConfigurationMode : ApplyAndMonitor

ConfigurationModeFrequencyMins : 15

Credential :

DebugMode : {NONE}

DownloadManagerCustomData :

DownloadManagerName :

LCMCompatibleVersions : {1.0, 2.0}

LCMState : Idle

LCMStateDetail :

LCMVersion : 2.0

StatusRetentionTimeInDays : 10

SignatureValidationPolicy : NONE

SignatureValidations : {}

MaximumDownloadSizeMB : 500

PartialConfigurations :

RebootNodeIfNeeded : False

RefreshFrequencyMins : 30

RefreshMode : PUSH

ReportManagers : {}

ResourceModuleManagers : {}

PSComputerName : W16TGT01

PSComputerName : W16TGT01

You can also run Get-DscLocalConfigurationManager locally if required.

#### Note

You can use the machine name with the -CIMSession parameter if accessing the target machine once. We’re using a CIM session here because we’ll be accessing the machine multiple times.

The most important point to note is the RefreshMode property. By default, it’s set to PUSH. This makes it easy to push configurations to a machine, but it means you have to do some work to convert to a pull environment.

Most of the properties are self-explanatory. Documentation for the LCM properties can be found at <http://mng.bz/e7n8>. We need to call out a few of the properties.

ConfigurationModeFrequencyMins controls how often (in minutes) the current configuration is checked and applied. It’s ignored if ConfigurationMode is set to ApplyOnly.

RefreshFrequencyMins controls how often (in minutes) the LCM checks a pull server to get updated configurations. This is ignored if LCM isn’t configured to use a pull server.

#### Note

ConfigurationModeFrequencyMins must be a multiple of RefreshFrequency-Mins, or RefreshFrequencyMins must be a multiple of Configuration-Mode-FrequencyMins.

ConfigurationMode has a number of possible settings:

* ApplyOnly—DSC applies the configuration. No further action is taken until a new configuration is available.
* ApplyAndMonitor—Default value. In this mode, the LCM applies any new configurations, and if the configuration of the machine drifts from the desires state (due to a manual change possibly), the change is logged.
* ApplyAndAutoCorrect—New configurations are applied. Any drift in configuration is logged, and the current configuration is reapplied.
* RefreshMode—Can be PUSH (default) or PULL. If set to PULL, a pull server must be configured through the ConfigurationRepositoryWeb property.
* ConfigurationID—A GUID used to identify the machine to a pull server.

Now that you’ve been introduced to the LCM properties, let’s look at changing the LCM settings.

#### Changing LCM Settings

You use Get-DscLocalConfigurationManager to view the LCM settings, so it’s probably not a big surprise that you use Set-DscLocalConfigurationManager to change the LCM settings. You can’t use the cmdlet directly. You first need to provide a configuration.

#### Listing 18.9: Changing the LCM settings

[DSCLocalConfigurationManager()]

Configuration LCM {

Param (

[Parameter(Mandatory=$true)]

[string[]]$ComputerName

)

Node $Computername

{

Settings

{

ConfigurationMode = 'ApplyAndAutoCorrect'

RebootNodeIfNeeded = $true

}

}

}

LCM -computername W16TGT01 -OutputPath .\MOF

The first thing to notice is the [DSCLocalConfigurationManager()] decorator. This is required if you’re working with the LCM—otherwise, DSC will assume you’re using a normal resource. A parameter block is used to supply the computer names to which the configuration will be applied.

The Settings resource is used to modify the basic LCM settings. You’ll see other options that can be used for controlling pull servers in the next section. In this example, we’re changing the LCM so that it automatically corrects configuration drift and allows the machine to reboot if needed during the application of a configuration.

Running the configuration produces a MOF file, as expected:

PS> C:\Scripts\Listing18.9.ps1

Directory: C:\scripts\MOF

Mode LastWriteTime Length Name

---- ------------- ------ ----

-a---- 13/02/2017 12:48 1166 W16TGT01.meta.mof

But notice the name of the MOF file: W16TGT01.meta.mof. The .meta.mof extension is used to differentiate MOF files used to configure the LCM from standard configuration MOF files used to configure the server.

#### Note

This file is commonly known as the metaconfiguration because it contains information that configures the local configuration manager.

Set-DscLocalConfigurationManager applies the LCM configuration rather than Start-DscConfiguration:

PS> Set-DscLocalConfigurationManager -CimSession $cs -Path .\MOF\ -Verbose

VERBOSE: Performing the operation "Start-DscConfiguration:

SendMetaConfigurationApply" on target "MSFT\_DSCLocalConfigurationManager".

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters,

''methodName' = SendMetaConfigurationApply,

'className' = MSFT\_DSCLocalConfigurationManager,

'namespaceName' = root/Microsoft/Windows/DesiredStateConfiguration'.

VERBOSE: An LCM method call arrived from computer W16TGT01 with user sid S-1-

5-21-759617655-3516038109-1479587680-1104.

VERBOSE: [W16TGT01]: LCM: [ Start Set ]

VERBOSE: [W16TGT01]: LCM: [ Start Resource ] [MSFT\_DSCMetaConfiguration]

VERBOSE: [W16TGT01]: LCM: [ Start Set ] [MSFT\_DSCMetaConfiguration]

VERBOSE: [W16TGT01]: LCM: [ End Set ] [MSFT\_DSCMetaConfiguration]

in 0.0470 seconds.

VERBOSE: [W16TGT01]: LCM: [ End Resource ] [MSFT\_DSCMetaConfiguration]

VERBOSE: [W16TGT01]: LCM: [ End Set ]

VERBOSE: [W16TGT01]: LCM: [ End Set ] in 0.7620 seconds.

VERBOSE: Operation 'Invoke CimMethod' complete.

VERBOSE: Set-DscLocalConfigurationManager finished in 1.105 seconds.

Let’s check our change:

PS> Get-DscLocalConfigurationManager -CimSession $cs |

select ConfigurationMode, RebootNodeIfNeeded

ConfigurationMode RebootNodeIfNeeded

----------------- ------------------

ApplyAndAutoCorrect True

and our target machine’s LCM shows the settings we desire.

Now that you know how to modify the LCM settings, it’s time to discover how to configure the LCM to use a pull server.

### 18.5.2 Configuring LCM to Use a Pull Server

Configuring a machine’s LCM to use a pull server involves modifying the LCM settings and providing the data the LCM needs to find the pull server. An example configuration to enable the use of the pull server we created earlier is shown in the following listing.

#### Listing 18.10: Configuring LCM to use the pull server

[DSCLocalConfigurationManager()]

Configuration LCMpull {

param (

[Parameter(Mandatory=$true)]

[string[]]$ComputerName,

[Parameter(Mandatory=$true)]

[string]$guid,

[Parameter(Mandatory=$true)]

[string]$ThumbPrint

)

Node $ComputerName {

Settings { 1

AllowModuleOverwrite = $True

ConfigurationMode = 'ApplyAndAutoCorrect'

RefreshMode = 'Pull'

ConfigurationID = $guid

}

ConfigurationRepositoryWeb DSCHTTPS { 2

ServerURL = 'https://W16DSC02:8080/PSDSCPullServer.svc'

CertificateID = $thumbprint

AllowUnsecureConnection = $false

}

ReportServerWeb RepSrv { 3

ServerURL = 'http://W16DSC02:9080/PSDSCPullServer.svc'

CertificateID = 'AllowUnencryptedTraffic'

AllowUnsecureConnection = $true

}

}

}

#$guid = New-Guid | select -ExpandProperty Guid 4

$guid = '5827c542-20bb-487c-89cb-484cbe5f0b1f'

$thumbprint=Invoke-Command -Computername W16DSC02 { 5

Get-Childitem Cert:\LocalMachine\My |

where Subject -Like 'CN=W16DSC02\*' |

Select-Object -ExpandProperty ThumbPrint}

LCMpull -computername W16TGT01 -Guid $guid ` 6

-Thumbprint $thumbprint -OutputPath .\MOF

* 1 LCM settings
* 2 Pull server settings
* 3 Report server settings
* 4 Setting GUID
* 5 Certificate thumbprint
* 6 Create MOF

The configuration starts with the [DSCLocalConfigurationManager()] decorator to ensure it targets the LCM. The important changes to the LCM settings 1 are to change the RefreshMode to PULL and to supply the GUID that’ll be used for the ConfigurationId.

The pull server configuration 2 includes the URL of the server and its certificate thumbprint. The reporting server 3 is configured to use unencrypted traffic (HTTP instead of HTTPS). In a production environment, you’ll want to encrypt all traffic.

In this case, the GUID 4 for the ConfigurationId is supplied. The alternate option is to generate a new GUID.

#### Note

Configuring the target node or creating the first configuration to be pulled is a chicken-and-egg scenario: which is first? The correct answer is whichever works for the problem you’re trying to solve.

The pull server’s certificate thumbprint 5 can be retrieved directly from the pull server. Running the configuration 6 produces a .meta.mof file.

The LCM modifications are applied:

PS> Set-DscLocalConfigurationManager -ComputerName W16TGT01`

-Path .\MOF\ -Verbose

Testing the LCM has to be done over a CIM session:

PS> Get-DscLocalConfigurationManager -CimSession $cs

ActionAfterReboot : ContinueConfiguration

AgentId : CBE8E714-C86B-11E6-841F-00155D36C90B

AllowModuleOverWrite : True

CertificateID :

ConfigurationDownloadManagers : {[ConfigurationRepositoryWeb]DSCHTTPS}

ConfigurationID : 5827c542-20bb-487c-89cb-484cbe5f0b1f

ConfigurationMode : ApplyAndAutoCorrect

ConfigurationModeFrequencyMins : 15

Credential :

DebugMode : {NONE}

DownloadManagerCustomData :

DownloadManagerName :

LCMCompatibleVersions : {1.0, 2.0}

LCMState : Idle

LCMStateDetail :

LCMVersion : 2.0

StatusRetentionTimeInDays : 10

SignatureValidationPolicy : NONE

SignatureValidations : {}

MaximumDownloadSizeMB : 500

PartialConfigurations :

RebootNodeIfNeeded : False

RefreshFrequencyMins : 30

RefreshMode : Pull

ReportManagers : {[ReportServerWeb]RepSrv}

ResourceModuleManagers : {}

PSComputerName : W16TGT01

PSComputerName : W16TGT01

The machine is now configured to use the pull server.

You can either wait for the DSC refresh cycle to pull the configuration, or if you’re impatient, you can force a refresh:

PS> Update-DscConfiguration -CimSession $cs -Verbose -Wait

As always you should test the configuration:

PS> Test-DscConfiguration -CimSession $cs

True

You can also test that the share exists:

PS> Get-SmbShare -CimSession $cs

Name ScopeName Path Description PSComputerName

---- --------- ---- ----------- --------------

ADMIN$ \* C:\Windows Remote Admin W16TGT01

C$ \* C:\ Default share W16TGT01

IPC$ \* Remote IPC W16TGT01

Standard \* C:\TestFolder This is a test SMB Share W16TGT01

And that the file can be accessed:

PS> Get-Content -Path "\\W16TGT01\Standard\TestFile1.txt"

My first Configuration

So far, you’ve created configurations as a complete unit. In some circumstances, you may need to adopt a more granular approach—which leads us to the use of partial configurations.

## 18.6 Partial Configurations

Partial configurations were introduced in PowerShell v5. They allow you to deliver fragments of the configuration to your target rather than a complete configuration. The LCM on the target machine will combine the fragments before applying as a single configuration. In this section, we’ll examine your options for using partial configurations, when you should use them, and, possibly more importantly, when you shouldn’t. We’ll close with an example.

### 18.6.1 Partial Configurations: Yes or No

In this section, we’ll examine the reasons why you might want to use partial configurations and issues you may encounter when using partial configurations.

#### Partial Configuration Use Case

Why would you want to use partial configurations? Isn’t life complicated enough without splitting your configurations into a number of pieces?

The assumption behind partial configurations is that your environment isn’t managed by a single team (or single person, in smaller environments). You might have a team that manages the operating system on your servers, but other teams manage the applications, such as SQL Server, Exchange, or SharePoint. Alternatively, developers creating a new application may split the management of the configuration. In either case, a single configuration can’t be created to manage the target server due to permissions, skillset, or even office politics!

Partial configurations enable each team to create the configuration to manage their part of the environment. The server team creates a configuration to manage the operating system, and then other teams create configurations to manage the applications such as SQL Server, Exchange or your inhouse developed application. This also has the advantage of reducing the size of the individual configuration scripts.

The partial configurations can be pushed to the target machine, or the target can obtain the partial configurations from a pull server. If required, you can even use a mixture of push and pull modes to deliver the partial configurations.

Partial configurations could also be useful even if you have a team responsible for configuration work. You could use them to split up a large configuration and make it more manageable.

#### Issues with Partial Configurations

On the surface, partial configurations seem like an ideal solution to managing an environment with diverse responsibilities. But a number of potential pitfalls lie in wait. For example, does the server team’s standard settings for the server match the requirements of the application that’ll run on the server? You may find that both teams try to configure the same things in different ways and end up with errors.

There is a way to overcome these issues—it’s called communication. If you’re going to use partial configurations, you’re going to have to get the various people involved talking to each other. And you need someone with overall ownership of the whole configuration who can define the allowed configuration fragments.

Let’s see how partial configurations work, starting with push mode.

### 18.6.2 Pushing Partial Configurations

Using partial configurations in push mode is broadly similar to using standard configurations in push mode: you create the configurations and push them to the target. There are a few differences:

* You need to configure the LCM to accept the partial configurations.
* The partial configurations are pushed to the target using Publish-DSCConfiguration.
* The configuration is run using Start-DSCConfiguration.

This will be easier with an example. To keep it simple, we’ll only use built-in resources. Let’s use a configuration that has two tasks:

* Create an environmental variable and set its value
* Create a registry key

Here’s the configuration for creating an environmental variable.

#### Listing 18.11: Configuration to create environmental variable

Configuration EnvVarConfig {

param (

[string]$ComputerName

)

Import-DscResource -ModuleName PSDesiredStateConfiguration

Node $ComputerName {

Environment EnvironmentPC {

Ensure = 'Present'

Name = 'PCtestvar'

Value = 'PIA 3e'

}

}

}

EnvVarConfig -ComputerName W16CN01 -OutputPath .\MOF\Env\

The Environment resource is used to create an environmental variable called PCtestvar, which is given a value of 'PIA 3e'.

The important point is the output path that’s used when the configuration is run. You need to separate the MOFs for the partial configurations—otherwise they’ll overwrite each other because they have the same name. In practice, because the fragments are being created by different owners there will be no overlap if they’re created on different machines. This is only a problem when doing an example like this.

#### Note

If you ever find that one owner is authoring two fragments in production, you shouldn’t be using partial configurations.

In this case, we’ll create a subfolder in the MOF folder for the environmental variable configuration.

The next job is to generate the configuration that’ll create the registry key, as shown in the following listing.

#### Listing 18.12: Configuration to create the registry key

Configuration RegConfig {

param (

[string]$ComputerName

)

Import-DscResource -ModuleName PSDesiredStateConfiguration

Node $ComputerName {

Registry RegistryPC {

Ensure = 'Present'

Key = 'HKEY\_LOCAL\_MACHINE\SOFTWARE\RegTestKey'

Valuename = 'PCTestVar'

ValueData = 'PIA 3e'

ValueType = 'String'

}

}

}

RegConfig -ComputerName W16CN01 -OutputPath .\MOF\Reg\

String is the default registry type, but you should specify the data type for completeness and for debug purposes—and so that you’ll understand what you were trying to achieve when you look at the configuration in the future. A subfolder called Reg is used for the MOF file to ensure we don’t accidentally overwrite a partial configuration.

The last piece of the configuration is the control portion that defines the allowed partial configurations listing.

#### Listing 18.13: Control configuration

[DSCLocalConfigurationmanager()]

Configuration PCTest1 {

param (

[string]$ComputerName

)

Node $ComputerName {

PartialConfiguration EnvVarConfig {

Description = 'Sets the environmental variable'

RefreshMode = 'Push'

}

PartialConfiguration RegConfig {

Description = 'Sets the registry key'

RefreshMode = 'Push'

}

}

}

PCTest1 -ComputerName W16CN01 -OutputPath .\MOF

This will configure the LCM on the target machine, so it needs the [DSCLocalConfigurationmanager()] decorator. Each partial configuration needs to be listed in the control configuration using a block like this:

PartialConfiguration EnvVarConfig {

Description = 'Sets the environmental variable'

RefreshMode = 'Push'

}

The PartialConfiguration resource is used. The name that’s applied must match the configuration names used in the partial configuration scripts. A description helps explain what is happening.

You can apply the control configuration:

PS> Set-DscLocalConfigurationManager -Path .\MOF\ -ComputerName W16CN01

If you examine the LCM settings on the target machine

PS> $cs = New-CimSession -ComputerName W16CN01

PS> Get-DscLocalConfigurationManager -CimSession $cs

this line is of interest:

PartialConfigurations : {[PartialConfiguration]EnvVarConfig, [PartialConfiguration]RegConfig}

It shows the two partial configurations we want to apply. Let’s look at the PartialConfigurations setting in more detail:

PS> Get-DscLocalConfigurationManager -CimSession $cs |

select -ExpandProperty PartialConfigurations

ResourceId : [PartialConfiguration]EnvVarConfig

SourceInfo : C:\Scripts\Listing18.13.ps1::8::6::PartialConfiguration

ConfigurationSource :

DependsOn :

Description : Sets the environmental variable

ExclusiveResources :

RefreshMode : Push

ResourceModuleSource :

PSComputerName : W16CN01

ResourceId : [PartialConfiguration]RegConfig

SourceInfo : C:\Scripts\Listing18.13.ps1::13::6::PartialConfiguration

ConfigurationSource :

DependsOn :

Description : Sets the registry key

ExclusiveResources :

RefreshMode : Push

ResourceModuleSource :

PSComputerName : W16CN01

You can see the SourceInfo is set to the script containing the control configuration. The description and name of each partial configuration are also stored.

The next step is to publish the MOF files to the target machine:

PS> Publish-DscConfiguration -Path .\MOF\Env\ `

-ComputerName W16CN01 -Verbose

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters,

''methodName' = SendConfiguration,'className' =

MSFT\_DSCLocalConfigurationManager,'namespaceName' = root/Microsoft/Windows/

DesiredStateConfiguration'.

VERBOSE: An LCM method call arrived from computer W16DSC01 with user sid S-1-

5-21-759617655-3516038109-1479587680-1104.

VERBOSE: [W16CN01]: LCM: [ Start Set ]

VERBOSE: [W16CN01]: LCM: [ End Set ] Saved configuration

document into the partial configuration store.

VERBOSE: [W16CN01]: LCM: [ End Set ]

VERBOSE: Operation 'Invoke CimMethod' complete.

VERBOSE: Publish-DscConfiguration finished in 0.213 seconds.

PS> Publish-DscConfiguration -Path .\MOF\Reg\ -ComputerName W16CN01 -Verbose

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters,

''methodName' = SendConfiguration,'className' =

MSFT\_DSCLocalConfigurationManager,'namespaceName' = root/Microsoft/Windows/

DesiredStateConfiguration'.

VERBOSE: An LCM method call arrived from computer W16DSC01 with user

sid S-1-5-21-759617655-3516038109-1479587680-1104.

VERBOSE: [W16CN01]: LCM: [ Start Set ]

VERBOSE: [W16CN01]: LCM: [ End Set ] Saved configuration

document into the partial configuration store.

VERBOSE: [W16CN01]: LCM: [ End Set ]

VERBOSE: Operation 'Invoke CimMethod' complete.

VERBOSE: Publish-DscConfiguration finished in 0.11 seconds.

Because the LCM on the target machine is expecting the partial configurations, they don’t overwrite, as would normally happen if you sent multiple MOF files with the same name to the target.

Now it’s time to apply the configuration. The -UseExisting parameter on Start-DSCConfiguration tells the LCM to use the configurations it already has rather than push a new configuration to the target machine:

PS> Start-DscConfiguration -ComputerName W16CN01 `

-UseExisting -Wait -Verbose

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters,

''methodName' = ApplyConfiguration,'className' =

MSFT\_DSCLocalConfigurationManager,'namespaceName' = root/Microsoft/Windows/

DesiredStateConfiguration'.

VERBOSE: An LCM method call arrived from computer W16DSC01 with

user sid S-1-5-21-759617655-3516038109-1479587680-1104.

VERBOSE: [W16CN01]: [] Starting consistency engine.

VERBOSE: [W16CN01]: LCM: [ Start Resource ] [[Environment]EnvironmentPC]

VERBOSE: [W16CN01]: LCM: [ Start Test ] [[Environment]EnvironmentPC]

VERBOSE: [W16CN01]: [[Environment]EnvironmentPC]

(NOT FOUND) Environment variable 'PCtestvar'

VERBOSE: [W16CN01]: LCM: [ End Test ] [[Environment]EnvironmentPC]

in 0.3120 seconds.

VERBOSE: [W16CN01]: LCM: [ Start Set ] [[Environment]EnvironmentPC]

VERBOSE: [W16CN01]: [[Environment]EnvironmentPC]

(CREATE) Environment variable 'PCtestvar' with value 'PIA 3e

VERBOSE: [W16CN01]: LCM: [ End Set ] [[Environment]EnvironmentPC]

in 0.2030 seconds.

VERBOSE: [W16CN01]: LCM: [ End Resource ] [[Environment]EnvironmentPC]

VERBOSE: [W16CN01]: LCM: [ Start Resource ] [[Registry]RegistryPC]

VERBOSE: [W16CN01]: LCM: [ Start Test ] [[Registry]RegistryPC]

VERBOSE: [W16CN01]: [[Registry]RegistryPC] Registry

key 'HKLM:\SOFTWARE\RegTestKey' does not exist

VERBOSE: [W16CN01]: LCM: [ End Test ] [[Registry]RegistryPC] in

0.3750 seconds.

VERBOSE: [W16CN01]: LCM: [ Start Set ] [[Registry]RegistryPC]

VERBOSE: [W16CN01]: [[Registry]RegistryPC] (SET)

Create registry key 'HKLM:\SOFTWARE\RegTestKey'

VERBOSE: [W16CN01]: [[Registry]RegistryPC] (SET)

Set registry key value 'HKLM:\SOFTWARE\RegTestKey\PCTestVar'

'PIA 3e' of type 'String'

VERBOSE: [W16CN01]: LCM: [ End Set ] [[Registry]RegistryPC] in

0.3440 seconds.

VERBOSE: [W16CN01]: LCM: [ End Resource ] [[Registry]RegistryPC]

VERBOSE: [W16CN01]: [] Consistency check completed.

VERBOSE: Operation 'Invoke CimMethod' complete.

VERBOSE: Time taken for configuration job to complete is 2.841 seconds

Finally, test that the configuration worked:

PS> Test-DscConfiguration -ComputerName W16CN01

True

Partial configurations in push mode are more complicated than pushing a single, large configuration, but may be useful if you need to split your configurations to control their size. Production environments are more likely to be using a pull server, so we’ll see how partial configurations work with a pull server next.

### 18.6.3 Pulling Partial Configurations

A pull server offers scalability for a production environment compared to using push mode, but partial configurations bring additional complexity.

To configure a target machine using partial configurations in pull mode you need to

* Modify the LCM of the target machine to use pull mode and tell it which partial configurations to use
* Create the configurations, rename them, and copy to the pull server
* Wait for the target machine refresh cycle to apply the configurations or force an immediate refresh cycle

The first job is to modify the LCM. Use [listing 18.10](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-185-140.xhtml#ch18ex10) to originally configure the target machine to use a pull server. That listing can be modified, as shown in the following listing, to also include the definition of the partial configurations to use.

#### Listing 18.14: Modifying the LCM to use partial configurations in pull mode

[DSCLocalConfigurationManager()]

Configuration LCMpull {

param (

[Parameter(Mandatory=$true)]

[string[]]$ComputerName,

[Parameter(Mandatory=$true)]

[string]$guid,

[Parameter(Mandatory=$true)]

[string]$ThumbPrint

)

Node $ComputerName {

Settings {

AllowModuleOverwrite = $True

ConfigurationMode = 'ApplyAndAutoCorrect'

RefreshMode = 'Pull'

ConfigurationID = $guid

}

ConfigurationRepositoryWeb DSCHTTPS {

ServerURL = 'https://W16DSC02:8080/PSDSCPullServer.svc'

CertificateID = $thumbprint

AllowUnsecureConnection = $false

}

ReportServerWeb RepSrv {

ServerURL = 'http://W16DSC02:9080/PSDSCPullServer.svc'

CertificateID = 'AllowUnencryptedTraffic'

AllowUnsecureConnection = $true

}

PartialConfiguration EnvVarConfig { 1

Description = 'Sets the environmental variable'

ConfigurationSource = '[ConfigurationRepositoryWeb]DSCHTTPS'

RefreshMode = 'Pull'

}

PartialConfiguration RegConfig { 2

Description = 'Sets the registry key'

ConfigurationSource = '[ConfigurationRepositoryWeb]DSCHTTPS'

RefreshMode = 'Pull'

}

}

}

#$guid = New-Guid | select -ExpandProperty Guid

$guid = '5827c542-20bb-487c-89cb-484cbe5f0b1f'

$thumbprint=Invoke-Command -Computername W16DSC02 {

Get-Childitem Cert:\LocalMachine\My |

where Subject -Like 'CN=W16DSC02\*' |

Select-Object -ExpandProperty ThumbPrint}

LCMpull -computername W16TGT01 -Guid $guid `

-Thumbprint $thumbprint -OutputPath .\MOF

* 1 Environmental variable partial configuration
* 2 Registry key partial configuration

The changes involve adding the partial configuration definitions. The environmental variable configuration 1 uses the name of the configuration (exactly as we did for partial configurations in push mode). The configuration information includes a description, a reference to the pull server to be used, and the refresh mode—in this case, push.

A similar partial configuration definition is used for the registry key 2. Both partial configuration definitions are based on the name of the configuration. You’ll need those names in a moment.

You need to push the LCM configuration to the target machine:

PS> Set-DscLocalConfigurationManager -Path .\MOF\ `

-ComputerName W16TGT01 -Force

You can view the configuration over the same CIM session:

PS> Get-DscLocalConfigurationManager -CimSession W16TGT01 |

Format-List ConfigurationDownloadManagers, ConfigurationID, ConfigurationMode, PartialConfigurations, RefreshMode

ConfigurationDownloadManagers : {[ConfigurationRepositoryWeb]DSCHTTPS}

ConfigurationID : 5827c542-20bb-487c-89cb-484cbe5f0b1f

ConfigurationMode : ApplyAndAutoCorrect

PartialConfigurations : {[PartialConfiguration]EnvVarConfig,

[PartialConfiguration]RegConfig}

RefreshMode : Pull

Notice that the partial configurations are registered, and the refresh mode is set to pull.

Now create the partial configurations. You can use the code from [listings 18.11](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-186-141.xhtml#ch18ex11) and [18.12](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-186-141.xhtml#ch18ex12). You’ll need to change the computer name when creating the MOF files:

PS> EnvVarConfig -ComputerName W16TGT01 -OutputPath .\MOF\Env\

PS> RegConfig -ComputerName W16TGT01 -OutputPath .\MOF\Reg\

If you remember, when we created a pull configuration we needed to rename the MOF file using the configuration ID of the target server.

#### Note

In PowerShell v5, the requirements for configuration ID were relaxed. It can now be any string. It doesn’t have to be a GUID. Semantically, it’s equivalent to a Role ID now. Originally it was supposed to be a NodeID, but people kept using it for roles, so the PowerShell team repurposed it and added a separate property to identify the node.

When you use partial configurations in pull mode, the naming convention is

<configuration name>.<configuration id>.mof

Let’s quickly work through the steps to get your partial configurations to the pull server. First, you need to get the configuration ID of the target server:

PS> $cid = Get-DscLocalConfigurationManager -CimSession $cs |

select -ExpandProperty ConfigurationID

then use the GUID to rename the MOF files:

PS> Rename-Item -Path C:\Scripts\MOF\Env\W16TGT01.mof `

-NewName "EnvVarConfig.$cid.mof"

PS> Rename-Item -Path C:\Scripts\MOF\Reg\W16TGT01.mof `

-NewName "RegConfig.$cid.mof"

Each MOF file needs to have a checksum file generated:

PS> New-DscChecksum -Path .\MOF\Env\EnvVarConfig.5827c542-20bb-487c-89cb-

484cbe5f0b1f.mof -Force

PS> New-DscChecksum -Path .\MOF\Reg\RegConfig.5827c542-20bb-487c-89cb-

484cbe5f0b1f.mof -Force

Then you can copy the MOF files and the checksum files to the pull server:

PS> Get-ChildItem -Path .\MOF\Env -Filter "\*$cid\*" | Copy-Item -Destination

'C:\Program Files\WindowsPowerShell\DscService\Configuration\'

-ToSession $s -Force

PS> Get-ChildItem -Path .\MOF\Reg -Filter "\*$cid\*" | Copy-Item -Destination

'C:\Program Files\WindowsPowerShell\DscService\Configuration\'

-ToSession $s -Force

The new configurations will be applied at the next refresh—or if you’re impatient (or in testing mode), you can force a refresh of the configuration:

PS> Update-DscConfiguration -ComputerName W16TGT01 -Wait -Verbose

Partial configurations introduce a level of complexity and extra management that you may not need. The only valid scenario for partial configurations is when the fragments of configuration are being supplied by separate area owners. If you have complex configurations, you could break them down into a set of composite resources and use a master configuration that references the composite resources rather than trying to use partial configuration. We recommend that you only use partial configurations if you have to.

## 18.7 Summary

* You can use DSC to manage the configuration of your server estate.
* DSC is declarative and idempotent.
* DSC is standards-based.
* A DSC configuration uses DSC resources to define the configuration parameters.
* DSC can work in push and pull modes.
* A DSC configuration can be parameterized to manage one machine or many machines.
* Configuration data can be separated from the topology of the environment.
* When using pull mode, the MOF file generated by the configuration must be renamed using the target machine’s configuration ID.
* You must also generate a checksum of the MOF file, for a configuration to be used in pull mode, and copy both files to the pull server.
* Partial configurations enable you to split your configuration, and they can be created by different teams.

The one part of DSC we haven’t shown you yet is how to create your own DSC resources. We’ll cover that in [chapter 19](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-19-143.xhtml#ch19), where we discuss another new feature in PowerShell v5: PowerShell classes.

## Chapter 19: Classes in PowerShell

### Overview

This chapter covers

* The basic ideas underlying classes in PowerShell
* PowerShell class and enumeration creation
* Detailed discussion of properties and methods in PowerShell classes
* Method overloading and inheritance
* Class initialization and construction
* DSC resources based on PowerShell classes

*Oh brave new world that has such people in it!  
Miranda in William Shakespeare’s The Tempest*

PowerShell has always been a .NET language in that it worked with and consumed the types in the .NET framework, but it was always a kind of second-class citizen compared to other .NET languages because you couldn’t create new types directly in PowerShell. This has been fixed in PowerShell v5, which now supports the ability to define new classes as well as extend existing .NET classes.

**Note**

The class keyword was reserved in the earliest versions of PowerShell with the intent that the team would eventually add this capability to PowerShell. It only took a little under 10 years to do it because the PowerShell team didn’t want to rush into something as important as this.

In this chapter, we’re going to look at what defining classes allows you to do as a PowerShell scripter/programmer. Also, one of the primary drivers for introducing classes in v5 was to make it easier to define DSC management resources. In the latter part of the chapter, we’ll look at how this is done.

## 19.1 Writing Classes in PowerShell

The ability to write classes in PowerShell was introduced in PowerShell v5. In this section, we’ll show you how to create and use methods and properties in PowerShell classes.

#### Note

If you’ve done any programming in C#, while reading this chapter you should notice that PowerShell class syntax is a close subset of the C# syntax. The things that are missing from the subset include interface definition, property getters/setters, and the const, private, protected, and internal member attributes. Also, the new and overload attributes are not supported because all class members in PowerShell are virtual. This subset was specifically chosen to balance language complexity against expressive power, aligning PowerShell to the feature set available in other popular dynamic languages such as Python and Ruby. On the flip side, if you don’t program in C#, then learning PowerShell classes will also help you to learn C#.

The addition of classes to PowerShell is something of a game changer; it means that you can now program in PowerShell with all the capabilities present in mainstream dynamic programming languages. Classes also add a new level of reliability to programming in PowerShell. Many of the new features allow PowerShell to check your code for errors statically—while you’re writing the code instead of waiting until runtime. To maximize your experience using classes in PowerShell, it’s recommended that you use a PowerShell-aware editor like the PowerShell Integrated Scripting Environment (PowerShell ISE) or Microsoft Visual Studio Code (a free open source editor from Microsoft.) These tools can show you code errors while you’re editing your programs.

Now let’s dive in and see what PowerShell classes have to offer. We’ll start our exploration by looking at simple classes that contain only data members—properties.

### 19.1.1 Using Properties in a PowerShell Class

All the way through this book you’ve been using properties on objects. Properties are data members on objects and are fundamental to how PowerShell performs selecting, sorting, and formatting work. In this section, you’ll see how to define properties in your own classes in PowerShell. Let’s start with the simplest possible example:

PS> class Point

{

$x

$y

}

In this example, you use the class keyword, followed by the name of the class and a list of variable names that are to be the properties of the class. You could also write it this way on a single line:

PS> class Point { $x; $y }

This example shows how, with a small amount of text, you can define your first class. This is about as simple as it can get. To create an instance of this class, use either New -Object (a mechanism introduced way back in v1)

PS> New-Object Point

x y

- -

or, preferably, the new() method introduced in v5:

PS> [Point]::new()

x y

- -

#### Note

For code written for version 5 or above, when explicitly creating an object instance, you’re much better off using the PowerShell v5 [type]::new() method. It’s significantly faster than using New-Object and it’s easier to get array arguments correct. (Try passing a single argument that’s an array to New -Object, and you’ll quickly figure out how hard that is to do.)

Now use Get-Member on the instance of the Point class to make sure everything is as expected:

PS> $p = [Point]::new()

PS> $p | Get-Member

TypeName: Point

Name MemberType Definition

---- ---------- ----------

Equals Method bool Equals(System.Object obj)

GetHashCode Method int GetHashCode()

GetType Method type GetType()

ToString Method string ToString()

x Property System.Object x {get;set;}

y Property System.Object y {get;set;}

You’ll see all the characteristics you’d expect from a regular .NET type. Along with the members you defined (x and y), you also see the default .NET members GetType(), ToString(), and so on. This happens because Point is a regular .NET type. PowerShell classes are full .NET classes, which allows them to participate fully in the .NET ecosystem.

#### Note

For C# users, even though PowerShell 5+ doesn’t support the getter/setter syntax from C#, data members in PowerShell classes are properties, not fields. At some point in the future, the getter/setter syntax will likely be added.

Okay, let’s move along and see what else you can do with class properties. In the output from the previous examples, the values of the x and y members were empty (null). That’s because they’re untyped members. Let’s update the class to add type constraints to the members. It doesn’t take a lot more work to do this:

PS> class Point

{

[int] $x

[int] $y

}

PS> [Point]::New()

x y

- -

0 0

Now that you’ve added a type constraint [int] to each of the members, when you print out the instance, the values are 0 (the default value for integers) rather than null as they were in the earlier example.

But what if you want to have a specific initial value? The following example shows how to do this:

PS> class Point {

[int] $x = 1

[int] $y = 2

}

It’s as simple as assigning an initial value to the member. (At this point, this may seem familiar—it’s the same syntax used to initialize function parameters.)

Now let’s look at a way to create and initialize an instance all in one step by using cast initialization. To do this, you take a hashtable and cast it into the desired type:

PS> $p = [Point] @{ x=1; y=2 }

PS> $p

x y

- -

1 2

When you print out the value of $p, you can see that 1 has been assigned to x and 2 has been assigned to y. This is a powerful technique because you can take unschematized data in the form of hashtables or PSObjects and convert it into strongly typed objects. Let’s create a second class, Square, to see how this works. The Square class looks like this:

PS> class Square {

[Point] $c1

[Point] $c2

}

It’s another simple class, but this time the members are typed as being of the Point class you defined earlier. Let’s use the cast constructor to create an instance of this out of nested hashtables:

PS> $sq = [square] @{c1 = @{x=1; y=2}; c2 = @{x=3; y="4"}}

PS> $sq.c1.x

1

PS> $sq.c2.y

4

In this example, the top-level hashtable had two members, c1 and c2, each of which was defined in terms of x and y. In the cast construction, the constructor walked through the nested hashtable converting each element to the desired type, including converting the string “4” to the number 4. The same thing can be done with, for example, JSON documents. The following string is equivalent to the hashtable from the previous example:

PS> $jstr = '{"c1": {"x": 1, "y": 2}, "c2": { "x": 3, "y": "4"}}'

Now let’s convert it first into PSObjects using ConvertFrom-JSON and then cast the result into a [Square]:

PS> $sq = [square] ($jstr | ConvertFrom-Json)

PS> $sq.c1.x

1

PS> $sq.c2.y

4

And again, the cast initialization works all the way down, converting each piece to the required type. What happens if something is wrong in one of the source elements? Let’s find out. You’ll change the c2 element in the data to have x and z instead of x and y. Here’s what happens:

PS> $jstr = '{"c1": {"x": 1, "y": 2}, "c2": { "x": 3, "z": "4"}}'

PS> [square] ($jstr | ConvertFrom-Json)

Cannot convert value "@{c1=; c2=}" to type "Square".

Error: "Cannot convert value "@{x=3; z=4}" to type "Point".

Error: "Cannot convert the "@{x=3; z=4}" value of type "System.Management.Automation.

PSCustomObject" to type "Point".""

At line:1 char:1

+ [square] ($jstr | ConvertFrom-Json)

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidArgument: (:) [], RuntimeException

+ FullyQualifiedErrorId : InvalidCastConstructorException

The cast fails, and you get a somewhat informative message indicating what went wrong in the conversion process. With PowerShell v5, if you need to validate a JSON document, you have to create a set of classes that represents the schema of the JSON document.

### 19.1.2 Class Member Attributes

Members in PowerShell classes can optionally have the keyword attributes: hidden and static. Let’s learn a bit about them.

#### The Hidden Attribute

The hidden attribute makes a member, well, hidden. This means you won’t ever see the member by default; Get-Member won’t show it. You can force it to be shown by using -Force. Why would you want to hide a member? Hidden is intended to be used on class members that are used internally by the class but aren’t part of the end-user (public) signature of the class. In essence, these members are private to the class.

#### Note

Why not make them private like they are in C#? Because in the compiled language world, the debugger is a separate program from the compiler that has special access to everything. In contrast, the debugger in PowerShell is PowerShell—a reentrant session of the interpreter that lets you inspect the system. Because it’s only PowerShell, anything that makes members inaccessible to PowerShell makes them inaccessible to the debugger (because it’s PowerShell). In effect, the hidden attribute is a compromise between completely public members and private members. You don’t see them unless you explicitly ask for them.

#### The Static Attribute

The static attribute allows you to define static members in a PowerShell class. You saw static members previously when we discussed method invocation. Now you’ll see how to create these members. Here’s an example showing a class with a static member:

PS> class myclass {

static $foo = 123

}

In this example, you can see that all you need to do to create a static member is to prefix the member with the keyword static. Now you can access this member as follows:

PS> [myclass]::foo

123

Because the property is static, there’s no need to create an instance of the object in order to access the member. You can have hidden static members too. That looks like the following:

PS> class myclass2 {

static $foo = 123

hidden static $bar = 3.14

}

As mentioned in the previous section, you can still access the member

PS> [myclass2]::bar

3.14

but when you use Get-Member to look at the static members on the class

PS> [myclass2] | Get-Member -Static -Type Properties

TypeName: myclass2

Name MemberType Definition

---- ---------- ----------

foo Property static System.Object foo {get;set;}

you don’t see the bar property. If you want to see it, you use the -Force parameter on Get-Member, which looks like this:

PS> [myclass2] | Get-Member -Static -Type Properties -Force

TypeName: myclass2

Name MemberType Definition

---- ---------- ----------

bar Property static System.Object bar {get;set;}

foo Property static System.Object foo {get;set;}

#### Member Validation Attributes

Along with the keyword attributes (hidden and static), you can use the data transformation and data translation attributes you’re familiar with from advanced function parameters (see [section 7.2.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-72-61.xhtml#ch07lev2sec12)) on class members. Let’s see an example of a class using these attributes. The following example uses the [ValidateRange()] and [ValidateSet()] attributes to constrain the allowed values on the class members:

PS> class ApartmentPets

{

[int]

[ValidateRange(1,88)]

$UnitNumber

[string]

[ValidateSet("cat", "dog", "bird")]

$Type

[int]

[ValidateRange(0,3)]

$Count

}

This class could be used to keep track of the type and number of pets in each apartment of a building. But this goes beyond merely keeping track; the attributes on the class members prevent an entry from containing more than three (or fewer than zero) pets as well as restricting the type of pet and ensuring that the apartment number is valid. You can create a valid instance of this class by casting a hashtable into an instance of [ApartmentPets]:

PS> [ApartmentPets] @{ UnitNumber = 22; Type = "cat"; Count = 2 }

UnitNumber Type Count

---------- ---- -----

22 cat 2

Running this code creates an instance for apartment unit 22, which has two cats. But let’s try increasing the count a bit:

PS> [ApartmentPets] @{ UnitNumber = 22; Type = "cat"; Count = 10 }

Cannot create object of type "ApartmentPets". The 10 argument is greater than

the maximum allowed range of 3. Supply an argument that is less than or

equal to 3 and then try the command again.

At line:1 char:1

+ [ApartmentPets] @{ UnitNumber = 22; Type = "cat"; Count = 10 }

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : InvalidArgument: (:) [], RuntimeException

+ FullyQualifiedErrorId : ObjectCreationError

Trying to create an entry with 10 cats, results in an exception being thrown instead of creating an invalid instance. This ability to constrain the allowed values for members in a class provides a powerful way to ensure that all the objects you’re dealing with have valid data.

That’s enough about data members for now. Let’s take a short detour to look at how enumeration types are defined in PowerShell.

### 19.1.3 PowerShell Enumerations

The .NET framework provides a user-definable data type related to classes called an enumeration (usually shortened to enum) that defines a closed set of named constant values. For example, there’s a predefined enum type in the .NET framework for the days of the week, which you can access using a number or a name:

PS> [System.DayOfWeek]0

Sunday

PS> [System.DayOfWeek] "Saturday"

Saturday

In either case, the string containing the name of the day of the week is returned. In practice, the underlying type for enums is Int32, allowing them to be cast to integers:

PS> [int] [System.DayOfWeek] "Saturday"

6

#### Note

C# supports both long and int for the underlying type for enums. PowerShell currently supports only int.

Finally, you can view the list of values in an enumeration:

PS> [enum]::GetNames([System.DayOfWeek]) -join ', '

Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday

Let’s look at how you can create your own enums in PowerShell. Use the enum keyword to define the start of the enum. Supply a name and the list of values, and it’s done:

PS> enum foo { one; two; three }

As before, you can access the enum using numerical values, a static member reference, or a cast:

PS> [foo] 0

one

PS> [foo] "one"

one

PS> [foo]::one

one

Note that we didn’t specify any values when we defined the enum. If no explicit values are provided, the compiler assigns integer values in order, starting at 0. Specifying explicit values looks like this:

PS> enum foo { one = 1; two = 2; three = 3 }

Note that the values must be constant, which means you can’t, for example, use variables as the value for an enum member. The constant values, however, can be in any order, don’t need to be consecutive, and don’t even need to be unique:

PS> enum foo { one = 3; two = 20; three = 3 }

If you try to use a value that isn’t part of the enum, you’ll get an error:

PS> [foo]5

Cannot convert value "5" to type "foo" due to enumeration values that are not

valid. Specify one of the following enumeration values and try again. The

possible enumeration values are "three,one,two".

At line:1 char:1

+ [foo]5

+ ~~~~~~

+ CategoryInfo : InvalidArgument: (:) [], RuntimeException

+ FullyQualifiedErrorId : UndefinedIntegerToEnum

Notice that you helpfully get a list of legal values. If you use a name that isn’t in the list of values, you’ll be ignored:

PS> [foo]::five

PS>

#### Flags Enumerations

Another way to use enumerations is as a bit field or set of flags, where each element of the enumeration represents a unique bit or flag. This is done by adding the [flags()] attribute to the enum definition, like so:

PS> [flags()] enum mybitfield {one = 0x1; two = 0x2; three = 0x4; all = 0x7}

This example defines three individual bits using hex values for each element, one per bit, and a fourth element, all, that’s the bitwise AND of all three bits. You can use this with casts as follows:

PS> [int] [mybitfield] "one,three"

5

PS> [int] [mybitfield] "one,two,three"

7

PS> [int] [mybitfield] "all"

7

#### Using Enums

Now that you know all about enums, you may be wondering where to use them. They’re typically used in functions and parameters, method parameters, and class properties. For example, the following function uses the [DayOfWeek] enum we looked at earlier:

PS> function foo {

param([dayofweek] $bf)

"$bf is day $([int] $bf) in the week"

}

This function takes a single parameter constrained to be the [DayOfWeek]. Let’s run it:

PS> foo tuesday

Tuesday is day 2 in the week

Running the function automatically converts the string “Tuesday” into the enumerated type. Casing the enum value, you get the corresponding number of the day in the week. You also get type checking for invalid values. If you pass in a month instead of a string, it errors out:

PS> foo september

foo : Cannot process argument transformation on parameter 'bf'. Cannot

convert value "september" to type "System.DayOfWeek". Error: "Unable to

match the identifier name september to a valid enumerator name. Specify

one

of the following enumerator names and try again:

**Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday"**

At line:1 char:5

+ foo september

+ ~~~~~~~~~

+ CategoryInfo : InvalidData: (:) [foo], ParameterBindingArgumentTransformationException

+ FullyQualifiedErrorId : ParameterArgumentTransformationError,foo

Notice that the error message contains a complete list of the valid values. Also, with IntelliSense in the PowerShell ISE or Visual Studio Code, you’ll see the values in a drop-down menu when entering your code.

We’re finished with enumerations so we can finally move on to methods. Let’s begin.

## 19.2 Methods in PowerShell Classes

At long last, we’re going to look at how to add methods (behaviors) to your classes. You’ll learn many new things, but all the material in the previous sections in this chapter still applies. For example, static and hidden apply to methods as well as properties. Before we get started, there’s some basic information you need to learn. It’s summarized in the next section.

### 19.2.1 Method Basics

Although methods in PowerShell classes are, in many ways, similar to PowerShell advanced functions, you must be aware of a number of important differences. When the PowerShell team was designing the class’ features, it wanted to facilitate building larger programs with PowerShell. To that end, they made the following changes to the way things work:

* If a method is to have a return value, the type of that value must be specified as part of the method signature—for example, [int]. If no value is to be returned, then the return type of the method must be [void].
* When returning a value, you must use the return statement. You can’t allow a value to be written to the pipeline. Any values that are emitted directly to the pipeline are discarded. Although this might occasionally feel inconvenient, enforcing formal returns eliminates a common source of errors where objects are unintentionally leaked into the output stream, contaminating the output.
* Within a method, a variable must explicitly be assigned a value before it can be used in the method body. Using an unassigned variable will result in a compile-time error.
* Methods use lexical scoping, which means that the only variables you can use in the method body are ones that are defined in the method body. If you want to use global or script scope variables, you must use scope-qualified names; for example, $global:myvar or $script:myvar.
* Class member variables must be referenced as $this.myVariable. Likewise, if a method wants to reference other methods in the class, then it must also prefix the name with $this, as in $this.mymethod(2, 3).
* Static members of the class, both properties and methods, are referenced using the class name and the :: operator, as in [myclass]::MyStaticProperty or [myclass]::MyMethod(2, 3).

Reading through that list is a bit dry, so in the next few sections we’ll look at practical examples that illustrate these principles, starting with static methods.

### 19.2.2 Static Methods

We’ll start with static methods because these are the simplest method type. You’ll see that they resemble PowerShell functions in many ways.

Like static properties, static methods don’t require an instance of the class to be able to use them. Here is an example PowerShell class containing a static method.

#### Listing 19.1: A static method in a PowerShell class

class utils {

static [int] Sum([int[]] $na){

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

foreach ($n in $na) {

$result += $n

}

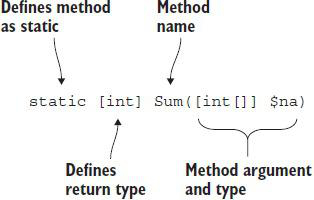
return $result

}

}

In this class, the Sum() method takes an integer array as its argument and returns the result of adding all of the values in the array together. Following the requirements spelled out in the previous section means that the syntax of a method declaration is a bit more complex than that of a function. Let’s break the signature into pieces, as shown in [figure 19.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19fig01).

Figure 19.1: Signature of a PowerShell class method



Looking at the code in [listing 19.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19ex01), you can see that the method is still similar to a function, but there are a couple of fundamental differences:

* The code defining the method is a scriptblock that has only an end block. You can’t use Begin, Process, and End blocks in methods.
* You must use the return keyword to return the results and exit the method. You can’t only emit the results to the pipeline the way you can with a PowerShell function.

Because Sum is a method, it’s invoked like other static methods you’ve seen throughout the book using static method invocation syntax:

PS> [utils]::Sum(1..10)

55

As a comparison, the code in [listing 19.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19ex01) can be converted to the more or less equivalent function as follows:

PS> function Sum {

param ([int[]] $na)

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

$result

}

foreach ($n in $na) {

$result += $n

}

$result

}

and you use the function version as normal:

PS> sum -na (1..10)

55

You now have two ways to create reusable pieces of code: static methods and functions. Both have their advantages: Functions provide the most natural command-line experience, whereas static methods are more programmer friendly and make it possible to write large PowerShell programs more reliably.

#### Note

There’s one other interesting advantage to static methods (at least at the time this book was written): Static method dispatch is orders of magnitude faster than function dispatch. Normally this doesn’t make much difference, but if you’re writing a tight, performance-sensitive loop that calls other code, you may consider writing that code as a static method rather than as a function. (In fact, method dispatch is fast for all methods, but static methods can most easily be used in place of functions, providing broader opportunities for performance enhancement in regular scripts.)

Now let’s look at instance methods, which require that you create an instance of the class before you can use them. That’s the topic of the next section.

### 19.2.3 Instance Methods

Let’s start by modifying [listing 19.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19ex01). First, you’ll rename the static Sum() method from [listing 19.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19ex01) to ISum(), indicating that it specifically sums integers. Then you’ll add a new method to perform the summation of an array of doubles.

#### Listing 19.2: Static and instance methods

class utils {

static [int] ISum([int[]] $na){

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

foreach ($n in $na) {

$result += $n

}

return $result

}

[double] DSum([double[]] $da){

$result = 0

if ($da -eq $null -or $da.Length -eq 0) {

return $result

}

foreach ($n in $da) {

$result += $n

}

return $result

}

}

The only change to the Sum() method was the name change. The big difference from [listing 19.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19ex01) is that you’ve added a second method, Dsum(), which sums an array of doubles. DSum() doesn’t have the static keyword and so is an instance method. The return type and the argument type are set to [double] and [double[]], respectively, and the variable $na is changed to $da—otherwise the code is the same as ISum().

Having created your class, you can use the static ISum() method as before:

PS> [utils]::ISum(1..10)

55

If you want to create an instance of the class, you can use the ::new() static method:

PS> $ui = [utils]::new()

#### Note

The new() method is a sort of extension method. It doesn’t exist on the object but is understood by the PowerShell language to mean that you want to create a new object. One consequence of this is that the new() method isn’t shown in the output of Get-Member.

Use the Get-Member cmdlet to view the class methods and properties for your new instance:

PS> $ui | Get-Member

TypeName: utils

Name MemberType Definition

---- ---------- ----------

DSum Method double DSum(double[] da)

Equals Method bool Equals(System.Object obj)

GetHashCode Method int GetHashCode()

GetType Method type GetType()

ToString Method string ToString()

Notice that the DSum() method is visible, but not the static ISum() method. If you want to see the static members of an object, you need to use the -Static parameter on Get-Member:

PS> $ui | Get-Member -Static

TypeName: utils

Name MemberType Definition

---- ---------- ----------

Equals Method static bool Equals(System.Object...

ISum Method static int ISum(int[] na)

ReferenceEquals Method static bool ReferenceEquals(Syst...

Having created the class instance, you can now use the method to sum a list of floating-point numbers:

PS> $ad = 1.1,2.2,3.3,4.4,5.5

PS> $ui.DSum($ad)

16.5

Our class has separate named methods defined for summing integers and doubles. Now let’s add properties to the class that will be used by the methods you’ve defined. You’ll add two new properties to the list: a static $ISumTotal property for the ISum() method and an instance property called $DSumTotal for the DSum() method. These properties will hold the running total of all of the summations.

#### Listing 19.3: Static and Instance methods with properties

class utils {

static [int] $ISumTotal = 0 1

static [int] ISum([int[]] $na){

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

foreach ($n in $na) {

$result += $n

}

[utils]::ISumTotal += $result

return $result

}

[double] $DSumTotal = 0.0 2

[double] DSum([double[]] $da){

$result = 0

if ($da -eq $null -or $da.Length -eq 0) {

return $result

}

foreach ($n in $da) {

$result += $n

}

$this.DSumTotal += $result

return $result

}

}

* 1 Static sum variable used by ISum()
* 2 nstance variable used by DSum()

In a static method, you reference the property 1 by using the name of the type, as in

[utils]::ISumTotal += $result

This isn’t new; it’s the same way you’ve always accessed static properties in a class. But the instance member introduces something new:

$this.DSumTotal += $result

Here you see a new variable, $this, being used to reference the specific instance 2 of the object in the method. You saw this before in [section 10.2.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-102-84.xhtml#ch10lev2sec6), with script methods. Script methods also have an automatic $this pointer to refer to the instance of the object that the method is accessing.

You’ve looked at static and simple instance methods. Now let’s look at the most sophisticated variation on instance methods. The PowerShell language allows for typed method parameters, which means that you can distinguish same-named methods by their list of parameters. This mechanism is called method overloading, which we’ll look at next.

#### Note

Obviously, PowerShell also allows type constraints on function parameters, but still you can’t have more than one function with the same name. For functions, the effective equivalent of overloads is parameter sets. There are advantages to both approaches. Overloading is expected and traditional for class methods. Parameter sets give you the expected and traditional commandline experience. The problem spaces are related but differ in significant ways, and so you have two different solutions.

### 19.2.4 Method Overloads

A method overload occurs when a method has two or more versions, which differ only by their input definitions. The combination of name plus input definitions is called the method signature. The next listing shows the utils class rewritten to utilize method overloads. Instead of having separate ISum() and DSum() methods, you can have two methods called Sum that have different signatures.

#### Listing 19.4: Using method overloads

class utils {

[int] Sum([int[]] $na){

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

foreach ($n in $na) {

$result += $n

}

return $result

}

[double] Sum([double[]] $na){

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

foreach ($n in $na) {

$result += $n

}

return $result

}

}

Instead of having the ISum() and DSum() methods as in [listing 19.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19ex03), this time you have two copies of the Sum() method, both of which are instance methods. They could have as easily been defined as two static methods, but you can’t have one be static and the other be instance. As we’ve discussed, the only difference between the two lies in the method signatures:

[int] Sum([int[]] $na)

[double] Sum([double[]] $na)

The first version has integer input and output. The second version works with doubles. Create an instance of the class so you can look at it:

PS> $ui = [utils]::new()

Using Get-Member, you see two signatures for the Sum method:

PS> $ui | Get-Member

TypeName: utils

Name MemberType Definition

---- ---------- ----------

Equals Method bool Equals(System.Object obj)

GetHashCode Method int GetHashCode()

GetType Method type GetType()

Sum Method int Sum(int[] na), double Sum(double[] na)

ToString Method string ToString()

You can also get the overloads for a method by using the method name without the parentheses:

PS> $ui.Sum

OverloadDefinitions

-------------------

int Sum(int[] na)

double Sum(double[] na)

From the perspective of the person using them, both methods are invoked with identical syntax but different parameter types. You start with summing an integer array, which looks like this:

PS> $ui.Sum(1..10)

55

Then you define and sum an array of doubles:

PS> $ad = 1.1,2.2,3.3,4.4,5.5

PS> $ui.Sum($ad)

16.5

The PowerShell runtime looks at the name of the method to get the available overloads and then compares the signature of each overload against the parameters to decide which one to invoke. When distinguishing overloads, the runtime does a best-match comparison. It’s entirely possible to have two overloads that would work (in fact, the [double] variant would be perfectly happy with integers and convert them to doubles). When looking at an overload, the runtime picks the one with the closest match. If you pass in integers, the integer signature method is the closet match, even though the double signature method would work.

The methods you’ve seen so far have been visible to Get-Member and the user. As was the case with properties, at times you may want to hide internal methods from your users.

### 19.2.5 Hidden Methods

As discussed in [section 19.1.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-191-144.xhtml#ch19lev2sec2), all PowerShell class members (both methods and properties) are public; private members are not supported. As was the case for properties, you can use the hidden keyword to create methods that are hidden from the default Get-Member results and PowerShell IntelliSense, even though the method’s still a public member of the class. Typically, hidden is used to hide implementation details of the class, as shown in the following listing.

#### Listing 19.5: Using hidden methods

class HasLogging

{

[int] Add($x, $y)

{

$this.Log("add $x $y")

return $x + $y

}

[int] Subtract($x, $y)

{

$this.Log("subtract $x $y") 1

return $x + $y

}

hidden [void] log($msg) 2

{

# logging code goes here

}

}

* 1 Call to hidden logging method
* 2 Hidden method used for logging

In this example class, a common method, log(), 1 is used by the two other methods in the class. This is an internal logging method used by that class and is not intended to be called directly. As a consequence, it’s marked hidden 2 so as not to clutter up the user’s experience with the class. Let’s construct an instance of the class and see what Get-Member returns:

PS> [haslogging]::new() | Get-Member

TypeName: HasLogging

Name MemberType Definition

---- ---------- ----------

Add Method int Add(System.Object x, System.Object y)

Equals Method bool Equals(System.Object obj)

GetHashCode Method int GetHashCode()

GetType Method type GetType()

Subtract Method int Subtract(System.Object x, System.Object y)

ToString Method string ToString()

As expected, the log() method isn’t shown in the output.

Speaking of constructors, this is something we haven’t discussed yet. The next section covers object construction in detail.

### 19.2.6 Constructors in PowerShell Classes

A constructor is code used by the runtime to initialize an instance of a class. The constructor can populate some or all of the properties of a class. If you don’t specify a constructor, a class automatically gets a default constructor that creates an instance with all properties set to their default values, as you saw in the earlier examples.

Let’s return to our apartment pets example class:

PS> class ApartmentPets

{

[int]

[ValidateRange(1,100)]

$UnitNumber

[string]

$Type

[int]

[ValidateRange(0,3)]

$Count

}

This class has only the default constructor it gets automatically. This means that you have to create an instance explicitly and then assign it to each member:

PS> $petEntry = [ApartmentPets]::new()

PS> $petEntry.Count = 2

PS> $petEntry.Type = "cat"

PS> $petEntry.UnitNumber = 7

The alternative is to use the cast constructor:

PS> [ApartmentPets] @{ UnitNumber = 22; Type = "cat"; Count = 2 }

This works great in many cases but doesn’t address all circumstances—like, say, you can have up to 3 cats but only 2 dogs or up to 10 fish. That logic can’t be captured using attributes on the members. Instead, you’ll have to write a constructor. You can add one or more constructors to your class, again overloaded by having different signatures. A constructor has the same name as the class, as shown next.

#### Listing 19.6: Using a non-default constructor

class ApartmentPets

{

[int]

$UnitNumber

[string]

$Type

[int]

$Count

ApartmentPets(){}

ApartmentPets([int] $UnitNumber, [string] $Type, [int] $Count) 1

{

if ($UnitNumber -lt 1 -or $UnitNumber -gt 100)

{

throw [InvalidOperationException]::new(

"Unit number $UnitNumber is invalid. Must be in range 1-100")

}

$maxPets = switch ($Type)

{

cat { 3; break }

dog { 2; break }

fish { 10; break }

default {

throw [InvalidOperationException]::new(

"The allowed pets are dogs, cats & fish. A $type is not

allowed")

}

}

if ($count -gt $maxPets)

{

throw [InvalidOperationException]::new(

"You are only allowed to have up to $maxPets pets of type $Type")

}

$this.Count = $Count

$this.Type = $Type

$this.UnitNumber = $UnitNumber

}

}

* 1 Constructor to populate object properties

The list of properties in the class is identical to [listing 19.5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19ex05). A constructor has the same name as the class and can take zero or more arguments in the parentheses 1. Arguments are separated by commas. The code to populate the properties is found between the braces, {}.

#### Note

As soon as you add an explicit constructor to the class, the class no longer has a default constructor. If you still want a default constructor, you’ll have to add it yourself; you no longer get it for free. That said, if you write a constructor like the one in [listing 19.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19ex06), you don’t want a default constructor because that would allow the class user to bypass the very checks you’re trying to enforce in the explicit constructor.

You have to use $this to refer to the property name within the constructor:

$this.Type = $Type

$this indicates you’re dealing with the current object. To create a new object you still use the new() pseudo-static method:

PS> [ApartmentPets]::new(22, 'cat', 2)

UnitNumber Type Count

---------- ---- -----

22 cat 2

Error handling during object construction is managed by the constructor:

PS> [ApartmentPets]::new(22, 'cat', 4)

You are only allowed to have up to 3 pets of type cat

At line:27 char:13

+ throw [InvalidOperationException]::new(

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : OperationStopped: (:) [], InvalidOperationException

+ FullyQualifiedErrorId : You are only allowed to have up to 3 pets of type cat

or

PS> [ApartmentPets]::new(22, 'parrot', 4)

The allowed pets are dogs, cats & fish. A parrot is not allowed

At line:21 char:17

+ throw [InvalidOperationException]::new(

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : OperationStopped: (:) [], InvalidOperationException

+ FullyQualifiedErrorId : The allowed pets are dogs, cats & fish. A parrot is not allowed

#### Using new() with .NET classes

You can also use the new() option directly with .NET classes. For instance:

PS> [datetime]::new(2017,12,25)

25 December 2017 00:00:00

In this case you’re using the first constructor listed at <http://mng.bz/uV9G>. The integers supplied to the constructor represent year, month, and day, respectively.

This is equivalent to using the -ArgumentList parameter on New-Object:

PS> New-Object -TypeName datetime -ArgumentList 2017, 12, 25

25 December 2017 00:00:00

You’ve seen how to create and use properties, methods, and constructors on a new class, but what about the case where you want to modify an existing class?

## 19.3 Extending Existing Classes

The act of creating a class that extends any existing class is called inheritance. The original class is known as the base class. The new class, known as the derived class, inherits all of the methods and properties of the base class.

#### Note

This section is included to complete the coverage of PowerShell classes. Class inheritance is a programmer topic and not something we’d expect many IT pros to use.

This topic is something that’s definitely best explained through examples. We’ll show you how to create a new class based on an inherited class, how to override the methods of the base class, and how to access the methods and constructor of the base class.

The first item is to create a derived class.

### 19.3.1 Creating a Derived Class

Before you can create a derived class, you need a base class. The following listing shows the base class we’ll use for these examples.

#### Listing 19.7: The base class

class utils {

[int] Sum([int[]] $na){

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

foreach ($n in $na) {

$result += $n

}

return $result

}

}

You’ve seen this code several times already. The class has one method, Sum(), that sums an array of integers. Assume that you also want a method that will sum doubles. You could add a method overload as you did in [section 19.2.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19lev2sec6), but in this case, we’ve decided that you need a new class.

Here’s how to derive a new class from your base class.

#### Listing 19.8: The derived class

class utils { 1

[int] Sum([int[]] $na){

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

foreach ($n in $na) {

$result += $n

}

return $result

}

} 1

class newutils : utils { 2

[double] Sum([double[]] $na){ 3

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

foreach ($n in $na) {

$result += $n

}

return $result

}

}

* 1 The base class
* 2 Derived class start
* 3 Method signature for derived class

The base class 1 is as shown in [listing 19.8](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-193-146.xhtml#ch19ex08) and earlier examples in the chapter. The new class’s relationship 2 to the base class is shown by the first line of the class definition:

class newutils : utils

The derived class name follows the class keyword. A colon followed by the base class name shows the class from which the new class is inheriting members. The version of the Sum() method that sums doubles 3 is part of the new class.

An instance of the new class is created in the same way as for any PowerShell class:

PS> $ui = [newutils]::new()

If you examine $ui with Get-Member

PS> $ui | Get-Member

TypeName: newutils

Name MemberType Definition

---- ---------- ----------

Equals Method bool Equals(System.Object obj)

GetHashCode Method int GetHashCode()

GetType Method type GetType()

Sum Method double Sum(double[] na), int Sum(int[] na)

ToString Method string ToString()

you’ll see both method overloads. In this case, both method overloads are available to the new class:

PS> $ad = 1.1, 2.2, 3.3, 4.4, 5.5

PS> $ui.Sum($ad)

16.5

PS> $ui.Sum(1..10)

55

In this example, you’ve extended the base class by providing an extra method overload. What about the situation where you want to override the method in the base class?

### 19.3.2 Overriding Members on the Base Class

In this example you’ll override the Sum() method of the base class. You override the method by creating a method in the derived class that has the same signature: name, return (output) type, and arguments as a method in the base class.

#### Listing 19.9: Overriding the base class

class utils { 1

[int] Sum([int[]] $na){

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

foreach ($n in $na) {

$result += $n

}

return $result

}

}

class newutils : utils { 2

[int] Sum([int[]] $na){

$result = 0

if ($na -eq $null -or $na.Length -eq 0) {

return $result

}

$result = 1

foreach ($n in $na) {

$result \*= $n 3

}

return $result

}

}

* 1 Base class
* 2 Derived class
* 3 Change to method

The base class 1 is as in the previous section. The derived class 2 has a method with the same signature as the base class. This means it will override, or replace, the method from the base class. The method in the derived class is different in that it calculates the product of the input array 3 rather than the sum. Notice that $result is set to 1 before the calculations start. Multiplying by zero gives zero!

Creating an instance of the derived class hasn’t changed, but when you examine the instance of the class, you’ll see that the Sum() method has only a single overload:

PS> $ui = [newutils]::new()

PS> $ui | Get-Member

TypeName: newutils

Name MemberType Definition

---- ---------- ----------

Equals Method bool Equals(System.Object obj)

GetHashCode Method int GetHashCode()

GetType Method type GetType()

Sum Method int Sum(int[] na)

ToString Method string ToString()

When you create an instance of the class and call the Sum() method, you now get the product of the numbers in the array rather than the sum:

PS> $ui = [newutils]::new()

PS> $ui.Sum(1..10)

3628800

You can still access the Sum() method in the base class if required.

#### Accessing the Base Class

You saw how to override a method in the base class in [listing 19.9](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-193-146.xhtml#ch19ex09). If you need to use the method in the base class, you can do so. When you create your derived class, add a call to the base class’s method:

PS> class newutils2 : utils {

[int] Sum([int[]] $na){

return ([utils]$this).Sum([int[]] $na)

}

}

You’re casting to the base class so that you can access the method. When you create an instance and use it, you get the sum of the array:

PS> $ui = [newutils2]::new()

PS> $ui.Sum(1..10)

55

### 19.3.3 Extending .NET Classes

Because PowerShell classes are full .NET classes, it’s possible for PowerShell classes to extend existing, compiled .NET classes.

#### Note

There are limitations on this. Because PowerShell classes don’t currently support the protected member keyword, it’s not possible to extend classes that require overriding protected members.

Here’s an example where you overload one of the classes in the PowerShell code base.

#### Listing 19.10: Inheriting from a .NET class

using namespace System.Management.Automation

class FixCase : ArgumentTransformationAttribute 1

{

[object] Transform( 2

[EngineIntrinsics] $engineIntrinsics,

[object] $inputData)

{

[string] $data = $inputData -as [string]

if (-not $data) {

throw [PSArgumentNullException]::new("inputData")

}

return $data.SubString(0,1).ToUpper() +

$data.Substring(1).ToLower()

}

}

function AutocapPet

{

param (

[FixCase()] 3

[string]

$petType

)

return $petType

}

* 1 Class that derives from an existing .NET class
* 2 Overload the abstract Transform() method on that class
* 3 Apply the attribute to a function parameter

The class that you’re overloading is ArgumentTransformationAttribute 1, which is the base class for the argument transformation attributes that can be applied to function parameters. Inheriting from this class requires us to introduce another new concept called an abstract method. Abstract methods are part of a class’s signature that implies that the class can’t be used directly. A class with an abstract method is called an abstract class. You can have abstract classes without any abstract methods, but there isn’t much point to that. Abstract methods must be overloaded in the derived class to provide a concrete implementation.

That’s what you’re doing here with the 2 Transform method. The concrete Transform method overloads the abstract method in the base class. [Listing 19.10](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-193-146.xhtml#ch19ex10) also defined the function AutocapPet 3 that uses the [FixCase()] attribute. Call this function with an all-lowercase argument:

PS> AutocapPet gEorGE

George

The result is returned correctly cased, which is rather handy.

## 19.4 Classes, Modules, Using, and Namespaces

Now you know a lot about classes, but you still need to see how they’re organized for use and reuse. The fundamental element of reuse is, as always, the PowerShell module. You’ll organize your classes into modules and then use those modules in your scripts. The difference comes in how you use those modules. This is where another significant difference with classes shows up.

Whereas most things in PowerShell are resolved at runtime, PowerShell classes are processed at compile time. When you want to get all the type-checking benefits that classes provide, particularly IntelliSense support, it’s necessary for PowerShell to know about classes ahead of runtime. Unfortunately, the usual way modules are referenced is the Import-Module cmdlet, which is a runtime thing. The environment knows nothing about the contents of a module until the Import-Module cmdlet is run, loading the module (and executing any code the module contains). This doesn’t work for classes. Instead, PowerShell v5 introduced a new keyword, using, that does a superset of the things that Import-Module does.

#### Note

The implication here is that in PowerShell v5 scripts, you should generally prefer using over Import-Module because it provides better semantics for importation. There are still cases where you’ll need to use Import-Module, like deciding which module to load at runtime, but those are fairly rare. For the most part, you should use using in scripts and modules targeting PowerShell v5.

### The Using Assembly Pattern

The using keyword has three basic forms. The first form allows you to reference an assembly in your script. This form looks like this:

using assembly <assemblyName>

So, for example, to use the Windows.Forms assembly in your script, you would specify

using assembly System.Windows.Forms

at the top of the file. This will cause the Windows.Forms assembly to be loaded when you run your script. Now, because using is processed at compile time instead of runtime, the PowerShell ISE can show you mistakes as you’re typing rather than waiting until runtime. If you type an incorrect name in the ISE, you’ll see the error indicated by a red squiggle under the invalid name, as shown in [figure 19.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-194-147.xhtml#ch19fig02).

Figure 19.2: The PowerShell ISE shows using assembly name errors while editing

https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/images/19fig02.jpg

Like many of the features associated with classes, this will help you catch errors while you’re writing your code rather than waiting until you run it.

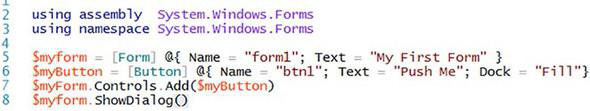
### The Using Namespace Pattern

The second variant for using is like this:

using namespace <namespace>

This variant will allow you to specify namespace prefixes that are used when PowerShell tries to resolve a type name. This can greatly simplify things when you have a long namespace, like System.Windows.Forms. [Figure 19.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-194-147.xhtml#ch19fig03) builds on the previous example, adding a namespace declaration for System.Windows.Forms, and then builds a small form example.

Figure 19.3: An example showing the use of using namespace to simplify using forms controls



In [figure 19.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-194-147.xhtml#ch19fig03), because you add the namespace directive, the code can be written to reference the type [Form] directly instead of as [System.Windows.Forms.Form]. Likewise, buttons can be referred to as [Button]. This certainly makes the code tidier.

### The Using Module Pattern

The final variant of the using directive is:

using module <moduleName>

This is the one you’ve been waiting for. It allows you to include a module in your script. Again, as with using assembly, the PowerShell ISE (or VSCode) will show an error (red squiggle) if the module name you specified to using module can’t be found.

The using keyword should always be used to load modules containing classes. That’s because classes are processed at compile time. All references to a class are resolved when the text referencing that class is scanned by the PowerShell parser. A class is visible only within the block of text defining it or in a block of text using a module with a defined class.

This block of text is called a compile unit. Let’s look at an example. You’ll take our friend the pets example from [listing 19.6](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-192-145.xhtml#ch19ex06) and move it into a module called apartmentpets.psm1. Now you can use using to reference this module in another script. You’ll create a new script called usingpets.ps1. This script will extend the class defined in apartmentpets to create a new class called apartmentpets2 and add a new member, Notes, to the class. This new class is shown in this listing.

#### Listing 19.11: apartmentpets2 class Inheriting from the apartmentpets class

using module apartmentpets 1

class apartmentpets2 : apartmentpets 2

{

[string]

$Notes

}

$apEntry = [apartmentpets2] @{ 3

Type = "dog"

Count = 1

UnitNumber = 66

Notes = "very friendly"

}

$apEntry | Format-List 4

* 1 Using apartmentpets brings the base class into the compile unit scope
* 2 The new class extends the existing one
* 3 Create an instance of the new type
* 4 Display that instance as a list

In this listing, you can see the using statement 1 that brings the base class, defined in the module apartmentpets, into the current compile unit. Then you create a new class 2 that extends the base class with a new member, $Notes. Once you have the new class defined, you can create an instance out of it using a cast initialization. You display it as a list 4. The output of this script looks like this:

PS> .\usingpets.ps1

Notes : very friendly

UnitNumber : 66

Type : dog

Count : 1

That’s exactly what you’d expect. But what happens if the using statement isn’t there and you put in an Import-Module instead? You’ll get the following error:

PS> .\usingpets.ps1

At C:\Users\bgpay\documents\usingpets.ps1:5 char:24

+ class apartmentpets2 : apartmentpets

+ ~~~~~~~~~~~~~

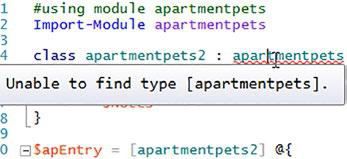
Unable to find type [apartmentpets].

+ CategoryInfo : ParserError: (:) [], ParseException

+ FullyQualifiedErrorId : TypeNotFound

This error occurs because you need to know the base type at compile time, and Import-Module doesn’t get called until runtime, when it’s too late. If you’re using the ISE, you’ll see an error as you’re typing, as shown in [figure 19.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-194-147.xhtml#ch19fig04).

Figure 19.4: The error message when the base class module is not imported with the using module statement



This should make it clear that you should always import modules containing classes with the using module statement.

### Using Modules and Namespaces

The last thing we need to talk about with using is how modules and namespaces interact. You were able to simplify the use of the Windows Forms classes with a using namespace statement. Why didn’t you need to do this with the module? Because when you use a module, the using module *<mymodule>* statement also has an implicit using namespace *<mymodule>* to simplify using the module. In practice, you could have written the new class as

using module apartmentpets

class apartmentpets2 : apartmentpets.apartmentpets

{

[string]

$Notes

}

but the implicit using namespace saves you the trouble and makes the most common scenario easier. To summarize: Every class defined in a module, lives in a namespace whose name corresponds to the module’s name. But because there is an implicit using namespace in the using module, you don’t have to worry about the namespace. The only time this will become a problem is when you import two different modules, m1 and m2, each of which contains a class with the same name, foo. In that case, you’d have to refer to the individual types using namespace-qualified names, as in [m1.foo] and [m2.foo].

At long last, we’ve finished our discussion of modules. But there’s one more point for the classes discussion. [Chapter 18](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-18-135.xhtml#ch18) promised to show you how much easier it was to write a DSC resource using classes rather than MOF and scripts. The last section of this chapter covers that topic.

## 19.5 Writing Class-Based DSC Resources

We said in [chapter 18](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-18-135.xhtml#ch18) that the easiest way to create your own DSC resource is to write it as a PowerShell class-based resource. Now that you know how to use PowerShell classes, it’s time we showed you how to write a DSC resource.

The process for creating class-based resources is:

1. Create a script module containing one or more class-based resources. This involves writing a class and annotating it with a specific set of attributes.
2. Copy the module to a directory in your module path, typically something like C:\Program Files\WindowsPowerShell\Modules\. Double-check to make sure that the module is in the right place. If it isn’t, you’ll get errors when trying to use it in a configuration. Again, smart editors like the ISE or VSCode will help you with this, calling out errors while you’re writing the configuration that uses the resource. It isn’t recommended that you put DSC modules in your personal module directory because that’s not available to the local configuration manager—the LCM runs as Local System.
3. Create a module manifest that loads the script module and exports the resources defined in the script module using the DscResourcesToExport module manifest member.
4. Import the module into your configuration script and use it like any other resource.

We’ll work through these steps in an example where you’ll create a DSC resource to control the Windows firewall; you’ll set it on or off for individual profiles.

The first step is to create the module with the class-based resource, as shown in the following listing. The class has a number of properties, corresponding to the resource properties, and three methods: Get(), Set(), and Test().

**Listing 19.12: Class-based DSC resource**

enum FWprofile { 1

Domain

Private

Public

}

enum Ensure {

Absent

Present

}

[DscResource()]

class FireWallStatus { 2

[DscProperty(Key)]

[FWprofile]$profileName

[DscProperty(Mandatory)]

[Ensure]$ensure

[DscProperty(NotConfigurable)]

[bool]$enabled

[FirewallStatus]Get() { 3

$fwp = Get-NetFirewallProfile -Name $this.profileName

$test = [Hashtable]::new()

$test.Add('ProfileName',$fwp.Name)

$test.Add('Ensure', $this.Ensure)

if ($fwp.Enabled) {$test.Add('Enabled', $true)}

else {$test.Add('Enabled',$false)}

return $test

}

[void]Set() { 4

$fwp = Get-NetFirewallProfile -Name $this.profileName

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

if (-not $fwp.Enabled) {

Set-NetFirewallProfile -Name $this.profileName -Enabled True

}

}

else {

if ($fwp.Enabled) {

Set-NetFirewallProfile -Name $this.profileName -Enabled False

}

}

}

[bool]Test() { 5

$fwp = Get-NetFirewallProfile -Name $this.profileName

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

if ($fwp.Enabled) {

return $true

}

else {

return $false

}

}

else {

if ($fwp.Enabled) {

return $true

}

else {

return $false

}

}

}

}

* 1 Enum definition
* 2 Start of class definition
* 3 Start of Get() method
* 4 Start of Set() method
* 5 Start of Test() method

The enums at the top of the listing 1 define the firewall profile names and the list of acceptable values for the Ensure option in the configuration (remember from [chapter 18](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-18-135.xhtml#ch18) that Ensure is set to Present to apply the configuration and Absent to remove the configuration).

The class definition 2 has a [DscResource()] decorator. This marks the class as a DSC resource. Your class must have this decorator if you want it to work as a DSC resource. The class properties have the [DscProperty()] decorator, so they’re also recognized by DSC. You’ll notice that DscProperty is modified for each property. The values have the following meanings:

* Key—This property is used to identify the instance to which the configuration will be applied. You must define a key property in a PowerShell class-based resource. In this case, it’s the firewall profile name.
* Mandatory—This property must have a value. In this case, Ensure controls whether the configuration is applied.
* NotConfigurable—This property isn’t configurable by the DSC resource. It’s populated in the Get() method to return additional information on the resource.

A PowerShell class-based resource must have three methods: Get(), Set(), and Test(). The class can have other helper methods if needed. The Get() 3 method returns an object showing the current state of the configuration. A hashtable is populated to be the return object. Notice that the Enabled nonconfigurable property is used to return the current state of the firewall profile. This method is called when Get-DscConfiguration is used.

The Set() method 4 is used by Start-DscConfiguration to apply the configuration. The method tests the current state of the configuration and enables or disables the firewall profile based on the combination of the values of Ensure and the current Enabled value.

Test() is the final method 5. This method returns a Boolean value to indicate whether the configuration of the firewall profile matches the desired configuration. Test-DscConfiguration uses this method.

Once written, the module is placed in C:\Program Files\WindowsPowerShell\Modules\ and a module manifest is created:

PS> New-ModuleManifest -Path 'C:\Program Files\WindowsPowerShell\Modules\

FirewallStatus\firewallstatus.psd1' `

-RootModule firewallstatus.psm1 -Guid ([GUID]::NewGuid()) `

-ModuleVersion 1.0 -Author 'Bruce & Richard' `

-Description 'Class based resource to toggle Windows firewall' `

-DscResourcesToExport 'FirewallStatus'

The -DscResourcesToExport parameter is used to create the list of resources available through the module. You must explicitly export the DSC resources from your module.

**Note**

The DscResourcesToExport module manifest member was added to improve the speed of resource discovery in a large set of modules. With this member, the resource discovery routines only need to scan the module manifest. Without it, the routines would have to scan all the .psm1 files, making the discovery process prohibitively slow. The down side is that if you forget to add it, your resource won’t be discovered, which can be hard to debug.

Your new DSC resource is now ready to use, so it’s time to create a configuration.

**Listing 19.13: Configuration using a class-based resource**

Configuration fwstatus {

param (

[Parameter(Mandatory=$true)]

[string[]]$computername,

[Parameter(Mandatory=$true)]

[string]$profilename,

[Parameter(Mandatory=$true)]

[bool]$enabled

)

Import-DscResource -ModuleName firewallstatus

if ($enabled) {$ens = 'Present'}

else {$ens = 'Absent'}

Node $computername {

FirewallStatus fwstoggle {

ProfileName = $profilename

Ensure = $ens

}

}

}

fwstatus -computername W16TGT01 -profilename Domain `

-enabled $true -OutputPath C:\Scripts\MOF

The script defines the configuration. The DSC resource is imported as usual. The value of Ensure is set based on the Boolean value of the configuration’s enabled parameter. You only need the profile name and Ensure to define the configuration for the node. The script runs the configuration to generate the MOF file.

Create a CIM session to the target computer and test the current setting of the firewall profile:

PS> $cs = New-CimSession -ComputerName W16TGT01

PS> Get-NetFirewallProfile -CimSession $cs | select Name, Enabled

Name Enabled

---- -------

Domain False

Private True

Public True

You’ve cheated and switched off the domain firewall profile. Use Start-DscConfiguration to apply your new configuration; the Set() method of your class performs the action of configuring the firewall.

PS> Start-DscConfiguration -CimSession $cs -Path .\MOF -Wait

You can determine the setting again:

PS> Get-NetFirewallProfile -CimSession $cs | select Name, Enabled

Name Enabled

---- -------

Domain True

Private True

Public True

The standard DSC cmdlets can be used to test the configuration (call the Test() method)

PS> Test-DscConfiguration -CimSession $cs

True

and get the current configuration (use the Get() method):

PS > Get-DscConfiguration -CimSession $cs

ConfigurationName : fwstatus

DependsOn :

ModuleName : FirewallStatus

ModuleVersion : 1.0

PsDscRunAsCredential :

ResourceId : [FireWallStatus]fwstoggle

SourceInfo :

enabled : True

ensure : Present

profileName : Domain

PSComputerName : W16TP5TGT01

CimClassName : FireWallStatus

Using PowerShell classes greatly simplifies the creation of DSC resources. This concludes our coverage of PowerShell classes.

## 19.6 Summary

* Classes can be written in PowerShell starting in version 5.0.
* Along with classes, PowerShell v5 or greater allows you to define your own enumerations (enums).
* Properties and methods in PowerShell classes can be static- or instance-based.
* All members of a PowerShell class are public, but members can be hidden from general users. Get-Member -Force will make hidden members visible.
* A method must use return rather than placing objects on the pipeline and must declare its return type. If it returns nothing, then its return type must be [void].
* Methods can be overloaded based on the types of their arguments—on their method signatures.
* Objects, both PowerShell and .NET classes, can be instantiated using New-Object or the ::new() pseudo-static method. For scripts targeting PowerShell v5 or higher, the use of ::new() is strongly recommended for performance and reliability reasons.
* PowerShell classes have a default constructor, but you can create additional constructors.
* PowerShell classes can inherit from .NET classes or other PowerShell classes.
* DSC resources can be created using PowerShell classes. These classes must be stored in modules and imported with the Import-DSCResource keyword like MOF-based resources.
* A class-based DSC resource must have proper annotations and implement Get(), Set(), and Test() methods.

So far, we’ve shown you how to use the features of the PowerShell language. In the next—and last—chapter, we’ll show you how to extend the way you use PowerShell through the use of the PowerShell APIs.

## Chapter 20: The PowerShell and Runspace APIs

### Overview

This chapter covers

* The PowerShell Application Programming Interface (API)
* How to perform isolated and concurrent operations
* Runspaces and runspace pools
* Out-of-process and remote runspaces
* Basic runspace management techniques

*Here’s a rule I recommend: never practice two vices at once.  
Tallulah Bankhead*

So far, we’ve been dealing with PowerShell as a shell and scripting environment. In this chapter, we’re going to look at it as an Application Programming Interface (API). An API is a set of functions, data structures, and classes that let you build applications on top of the software exposing that API. For example, the PowerShell ISE is an application that uses the PowerShell API. Normally, the PowerShell API is used by other programs for accessing PowerShell functionality, but it also turns out to be useful from within PowerShell itself. In effect, PowerShell scripts can act as host applications for other PowerShell engine instances, allowing you to perform advanced operations like dynamic pipeline construction, isolated execution, and concurrent operations.

## 20.1 PowerShell API Basics

In this section, we’ll look at the basic use patterns and structure of the PowerShell API. We’ll look at how to construct instances of the core API objects and how to compose those objects into executable pipelines.

The PowerShell API is accessed using the class System.Management.Automation.Power-Shell. That’s a bit long to type, so a type accelerator [PowerShell] is provided to simplify access to the class. This class provides a factory method Create() that creates instances of the [PowerShell] object.

#### Note

In object-oriented design, the factory method pattern is a way of constructing objects using a method instead of directly calling a specific type’s constructor. This abstracts the details of exactly which object is constructed and also allows the factory method to perform operations such as bookkeeping or object tracking before and after an object is created.

Once you have the [PowerShell] object instance, you can add commands to it using the AddCommand() method and finally invoke it using the Invoke() method. Let’s look at the simplest example using the API. You’re going to create an instance of the [PowerShell] object, add one command, Get-Date, to the object’s command collection, and then invoke it. This looks like the following:

PS> [PowerShell]::Create().AddCommand("Get-Date").Invoke()

19 May 2017 11:23:20

Take a look at this command and compare the English description to what you typed at the command line. Ignoring punctuation, they’re identical. The PowerShell API is an example of what’s known as a fluent API. A fluent API is one where the human-language representation and the code representation map one-to-one, item by item. This semantic mapping makes it easier for users to turn their intentions into executable code.

In the first example, the command you added, Get-Date, required no parameters, so you could add it and then invoke the command. Now let’s see how to handle a command that does take parameters. You’ll use the command Get-CimInstance with the argument Win32\_BIOS. In pure PowerShell, the command would be entered as

PS> Get-CimInstance -ClassName Win32\_BIOS

Using the PowerShell API, it looks like this:

PS> [PowerShell]::Create().AddCommand("Get-CimInstance"). `

AddParameter("ClassName", "Win32\_BIOS").Invoke()

SMBIOSBIOSVersion : 90.1380.768

Manufacturer : Microsoft Corporation

Name : 90.1380.768

SerialNumber : 004393254157

Version : MSFT - 0

Again, following the fluent API pattern, the method to add a parameter is AddParameter(). Now, suppose you only wanted to add a positional argument instead of the parameter name/value pair. As you can probably guess, the method to add an argument is AddArgument(). Here’s the same example but adding an argument instead of a parameter:

PS> [PowerShell]::Create().AddCommand("Get-CimInstance"). `

AddArgument("Win32\_BIOS").Invoke()

SMBIOSBIOSVersion : 90.1380.768

Manufacturer : Microsoft Corporation

Name : 90.1380.768

SerialNumber : 004393254157

Version : MSFT - 0

In this example, the parameter-binding logic figures out what parameter to bind the value Win32\_BIOS to, as it does in a PowerShell script.

Now that you have a basic understanding of the PowerShell API, commands, arguments, and parameters, let’s work on some more advanced examples.

### 20.1.1 Multi-Command Pipelines

So far, you’ve been working with only simple commands, but one of PowerShell’s greatest strengths is the ability to build pipelines of commands. In this section, we’re going to look at how to do that with the PowerShell API.

To create a pipeline with more than one command, all you need to do is to make a subsequent call to AddCommand() for each additional command you want to add to the pipeline. Each command you add becomes the next stage in the pipeline. Let’s see how this works with another example. In this example, you’re going to convert this pipeline

PS> Get-Process -Name Power\* | sort HandleCount -Descending

in PowerShell syntax into a [PowerShell] object. You start by creating the pipeline and adding parameters to it, as you’ve done previously:

[PowerShell]::Create().AddCommand("Get-Process").

AddParameter("Name", "Power\*").

Then you add the sort command along with its parameters to the pipeline. This requires a second call to AddCommand(), followed by calls to AddArgument() and AddParameter():

AddCommand("sort").

AddArgument("HandleCount").

AddParameter("Descending").

#### Note

Switch parameters can either be added using the AddParameter() overload that takes only a parameter name or by passing the parameter name along with a Boolean value.

Finally, you call Invoke() to cause the command to be executed. The complete command, equivalent to the original PowerShell expression, looks like this:

PS> [PowerShell]::Create().AddCommand("Get-Process"). `

AddParameter("Name", "Power\*"). `

AddCommand("sort"). `

AddArgument("HandleCount"). `

AddParameter("Descending"). `

Invoke()

Handles NPM(K) PM(K) WS(K) CPU(s) Id SI ProcessName

------- ------ ----- ----- ------ -- -- -----------

579 33 84744 62784 13.89 17484 1 powershell

481 29 62852 55724 14.33 36460 1 powershell

Again, because you’re using a fluent API, the transformation is pretty direct, with AddCommand() replacing the | pipe operator. This extends to as many stages in the pipeline as you need. Let’s look at a more complex example with four stages in the pipeline. This example counts the number of processes with more than 1000 handles. The PowerShell expression to do this is

PS> Get-Process | where HandleCount -GT 1000 |

Measure-Object | foreach Count

and the [PowerShell] API equivalent is

PS> [PowerShell]::Create(). `

AddCommand("Get-Process"). `

AddCommand("where"). `

AddArgument("HandleCount").AddParameter("GT").AddArgument(1000). `

AddCommand("Measure-Object"). `

AddCommand("foreach").AddArgument("Count"). `

Invoke()

18

The converted expression has four calls to AddCommand()—one for each stage in the pipeline.

#### Note

One thing to note in this example is that when specifying parameter names, you don’t need to specify the dash before the parameter name. The fact that you’re calling AddParameter() makes the intent clear.

### 20.1.2 Building Pipelines Incrementally

So far, all the examples have been showing the use of a single statement to create a [PowerShell] object. The fact that you can do this is one of the benefits of the fluent API design, but it’s not required. You could choose to build the pipeline incrementally across a series of statements. Let’s redo the final example in the last section. First, you need to get the [PowerShell] object into a variable:

PS> $p = [PowerShell]::Create()

Next, you add a command to that object:

PS> $p.AddCommand("Get-Process")

Commands : System.Management.Automation.PSCommand

Streams : System.Management.Automation.PSDataStreams

InstanceId : ffff110b-677a-4d72-9036-6f7d28d6803c

InvocationStateInfo : System.Management.Automation.PSInvocationStateInfo

IsNested : False

HadErrors : False

Runspace : System.Management.Automation.Runspaces.LocalRunspace

RunspacePool :

IsRunspaceOwner : True

HistoryString :

Wait—you get a whole bunch of output from this command, so clearly an object is being returned from the AddCommand() method! This is the same object you’ve stored in the variable $p. You haven’t seen this before because you’ve always been calling Invoke() at the end of your expressions. You can confirm that it’s the same object by adding another command to the object and comparing the return value to what’s stored in $p:

PS> $p -eq $p.AddCommand("where")

True

The result of the comparison shows that it always returns the same object. Next, you need to add the parameters and arguments to the object. Calls to AddParameter() and AddArgument() also return the same [PowerShell] object:

PS> $p -eq $p.AddArgument("HandleCount")

True

Now add the remaining parameters for the where command:

PS> $p = $p.AddParameter("GT").AddArgument(1000)

This time you’re assigning the result of the method calls back to $p. This is sensible because the object returned is the same as the object being assigned and it eliminates unnecessary objects in the output stream.

#### Note

In PowerShell scripts, expressions in statements return values that are placed in the output stream. To avoid getting objects you don’t want in the output stream, cast the expressions to [void] or assign the result to a variable. If you’re a C# programmer, this behavior would be unexpected because statements in C# discard any results that are explicitly consumed. Because the [PowerShell] API is used in both PowerShell and C#, it’s important to remember this difference in behavior when switching languages.

This is how the fluent API works: Each method call returns the original object so it can be used for the next method call. Now add the remaining commands from the example to the object in $p:

PS> $p=$p.AddCommand("Measure-Object").AddParameter("Sum"). `

AddCommand("foreach").AddArgument("Count")

The complete pipeline object is now available in $p ready to invoke. Let’s invoke it now:

PS > $p.Invoke()

18

Because the expression is still available in $p, you can invoke it again and again:

PS > $p.Invoke()

18

This way, you build the [PowerShell] object only once, regardless of how many times you need to invoke it.

The ability to incrementally build up pipelines is useful because you can do other processing or conditional logic between the steps to decide how to proceed. For example, within a script, you may want to dynamically add filters to the output of the script. [Listing 20.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-202-152.xhtml#ch20ex01) shows an enhanced file list example demonstrating how this can be useful. Feel free to skip forward and take a look, but for the next section, we’re going to switch to an extremely important topic we’ve glossed over so far: error handling.

### 20.1.3 Handling Execution Errors

So far, everything we’re tried has worked perfectly. But we live in an imperfect world and so need to look at how to deal with errors. Remember that PowerShell has two types of errors: terminating, which halt execution, and nonterminating, which are reported. In the simplest case, a terminating error that occurs when you invoke a [PowerShell] object will result in an exception being thrown. Let’s see what happens with a command-not-found error:

PS> [PowerShell]::Create().AddCommand("foobar").Invoke()

Exception calling "Invoke" with "0" argument(s): "The term 'foobar' is not

recognized as the name of a cmdlet, function, script file, or operable

program. Check the spelling of the name, or if a path was included, verify

that the path is correct and try again."

At line:1 char:1

+ [PowerShell]::Create().AddCommand("foobar").Invoke()

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : NotSpecified: (:) [], MethodInvocationException

+ FullyQualifiedErrorId : CommandNotFoundException

You can trap this exception using the try/catch statement (see [section 14.2.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-142-115.xhtml#ch14lev2sec5)) as follows:

PS> try { [PowerShell]::Create().AddCommand("foobar").Invoke() }

catch { "Caught exception: $\_" }

Caught exception: Exception calling "Invoke" with "0" argument(s): "The term

'foobar' 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."

This solution is good for terminating errors, but what about nonterminating errors that are written to the error stream? Let’s try this by writing an explicit error:

PS> [PowerShell]::Create().AddCommand("Write-Error"). `

AddArgument("An error").Invoke()

And you get … nothing? Correct—nothing because the command succeeded in that it wasn’t terminated, but there was still an error. Let’s see how you can handle this case. The most important point is to assign the [PowerShell] object to a variable before invoking it. That’s because the [PowerShell] object has a number of fields that you’ll need to examine after the invocation completes. Let’s set this up

PS> $p = [PowerShell]::Create().AddCommand("Write-Error"). `

AddArgument("An error")

and invoke the command

PS> $p.Invoke()

As in the earlier example, execution completes successfully with no indication of an error. Instead, you have to check properties on the [PowerShell] object. First, to see if any errors occurred, terminating or otherwise, you’ll check the HadErrors property. If any errors were generated during execution, terminating or otherwise, this property will be true. Let’s look:

PS> $p.HadErrors

True

Yes, an error did occur. Okay, you want to see what that error was. To do that, you need to look at the Streams property on the [PowerShell] object. The Streams property has one member for each of the streams PowerShell supports, as shown here:

PS> $p.Streams | Get-Member -Type Property | foreach Name

Debug

Error

Information

Progress

Verbose

Warning

Check the count on the Error stream, which, per the example, should contain one record:

PS> $p.Streams.Error.Count

1

and it does. Finally, we can dump out the error:

PS> $p.Streams.Error

Write-Error : An error

+ CategoryInfo : NotSpecified: (:) [Write-Error], WriteErrorException

+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,Microsoft.PowerShell.

Commands.WriteErrorCommand

As you might expect, anything written to the other streams during execution will be available in the respective stream property. Now create a [PowerShell] object that will emit a warning:

PS> $p = [PowerShell]::Create(). `

AddCommand("Write-Warning").AddArgument("A warning")

PS> $p.Invoke()

PS> $p.Streams.Warning

A warning

and, to check, examine the HadErrors property:

PS> $p.HadErrors

False

This confirms that no errors occurred during execution.

At this point, we’ve now looked at adding commands and parameters to [PowerShell] objects and how to handle errors with the PowerShell API. This completes our discussion of the basic use of the PowerShell API. In the next section, we’ll look at additional capabilities the API provides.

### 20.1.4 Adding Scripts and Statements

In the previous section we covered only the use of simple commands with the [PowerShell] object. In this section, we’ll look at two additional types of content you can add to the object: scripts and statements. Let’s start with scripts.

#### Adding Scripts to the Pipeline

We’ve looked at adding single commands, with or without parameters, and arguments to build pipelines using the PowerShell API. Now we’re going to look at another way of adding executable content to the [PowerShell] instance. As well as commands, the PowerShell API allows you to add scripts to an instance. As you might expect by now, this is done through the AddScript() method.

#### Note

Using the word script in this context is a bit confusing because scripts are technically commands. If you want to invoke a script named myscript.ps1, then you should call AddCommand("myscript.ps1") because myscript.ps1 is a simple command—it’s implemented as a script. The AddScript() method is about adding expressions to the pipeline, not commands. A much better name for the AddScript() method would have been AddExpression(), so in the same way that AddCommand() parallels Invoke-Command, AddExpression() would have paralleled Invoke-Expression. Unfortunately, no one thought of that at the time!

As always, we’ll start by looking at a basic example. You’re going to pass in a simple expression, 2+2, to be evaluated:

PS> [PowerShell]::Create().AddScript{2+2}.Invoke()

4

This example executes the expression 2+2. In many ways, AddScript() is the easiest method to use—just pass in the PowerShell code and execute it.

#### Note

This example showed passing a scriptblock to AddScript(). You can pass a string and get the same result. The advantage of using a scriptblock is that you get syntax checking on the code passed to the API when the object is created instead of deferring it to runtime.

Moving on, let’s try something more complex with a script that contains three statements that emit the numbers 1, 2, and 3:

PS> $p = [PowerShell]::Create().AddScript{1;2;3}

PS> $p.Invoke()

1

2

3

This gives you the expected response. Now let’s use a foreach loop to square these values:

PS> $p = [PowerShell]::Create(). `

AddScript{ foreach ($i in 1,2,3) { $i \* $i }}

PS> $p.Invoke()

1

4

9

This illustrates that you can use any PowerShell construct with AddScript(). Anything that can go in a scriptblock can be used with AddScript(). This implies that you can also deal with input in the script you’re adding to the [PowerShell] object. In a function or scriptblock, you can process input in two ways: by using $input in the end block or by creating a process block in the script. Both approaches work with AddScript(). By default, the script that’s passed to the AddScript() method is run as if it was the end block in the script. This means you can use $input to get the input from the pipeline:

PS> $p = [PowerShell]::Create(). `

AddCommand("Get-Process"). `

AddScript{ $input |

where { $\_.name -like "csr\*" } |

foreach name

}

PS > $p.Invoke()

csrss

csrss

This example takes the output of Get-Process, filters for processes matching "csr\*", and then returns the name of the process. Note that, because this is running in the end block, there’s no streaming. The prior command is run to completion before the added script is run. You can fix this by using a process block in the script. Create a new example that looks for process names in the process block:

PS> $p = [PowerShell]::Create(). `

AddCommand("Get-Process"). `

AddScript{process {

if ($\_.name -like "csr\*")

{

$\_.name

}

}

}

PS> $p.Invoke()

csrss

csrss

This example uses $\_ to get the current pipeline object so it doesn’t have to do any stream processing. Thus, a simple if statement is all that’s needed. Now try a script that both returns values and writes errors:

PS> $p = [PowerShell]::Create(). `

AddScript{ 1; Write-Error "@ is an error"; 3 }

PS> $p.Invoke()

1

3

Invoking the example returns the output of the first and third statements. You need to check the [PowerShell] object for the error. First, verify that the error occurred:

PS > $p.HadErrors

True

and then dump out the error itself:

PS > $p.Streams.Error

1; Write-Error "@ is an error"; 3 : @ is an error

+ CategoryInfo : NotSpecified: (:) [Write-Error], WriteErrorException

+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException

#### Note

This last example illustrates that just because the call to Invoke() returned a value doesn’t mean that there wasn’t an error. When using the [PowerShell] API, you should always check HadErrors and the streams to see if there were any errors.

We’ve now covered pretty much everything about scripts, so let’s look at the last type of content you can add to a [PowerShell] object.

#### Adding Statements to the Pipeline

The last element type that can be added to a [PowerShell] object is a “statement.” This term is in quotes because it doesn’t mean statement quite the same way as we do in PowerShell script; we’re not talking about if statements or while loops. What the AddStatement() does is add a second pipeline to the [PowerShell] object, resulting in a collection of pipelines that are executed one after the other. The output of all the pipelines/statements is aggregated and returned from the Invoke() method. This is easiest to understand with an example. You’re going to create a [PowerShell] object that has three statements, each of which is a script that returns the number corresponding to the statement. First, create the [PowerShell] object and store it in the variable $p:

PS> $p = [PowerShell]::Create()

Next, add the first script, which returns the value 1.

PS> $p = $p.AddScript{1}

Now call the AddStatement() method to indicate that you’re starting a new statement:

PS> $p = $p.AddStatement()

Now add the second script returning 2 as the content of the second statement:

PS> $p = $p.AddScript{2}

Finally, add the third statement and script in one step:

PS> $p = $p.AddStatement().AddScript(3)

You now have a complete object containing three statements that execute one after the other. Let’s call Invoke() and see the result:

PS> $p.Invoke()

1

2

3

As expected, you get the three numbers corresponding to each of the statements.

#### Adding Statements vs. Adding Scripts

Using the AddStatement() method may seem like an awkward way to execute multiple pipelines, particularly when you could call AddScript(). The primary scenario for AddStatement() is sending a series of commands to a remote runspace in NoLanguage mode, as described in [section 11.6.4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-116-99.xhtml#ch11lev2sec28). Runspaces in NoLanguage mode won’t accept script elements in the pipeline, so the only way to perform multiple actions in a single batch in this scenario is to use AddStatement().

So far, we’ve talked about runspaces in a fairly peripheral manner. In the next section, we’re going to look at them directly.

## 20.2 Runspaces and the PowerShell API

In this section, we’ll look at how runspaces, which are PowerShell engine instances, interact with the PowerShell API. A runspace is a container that holds everything needed to execute PowerShell code. This container holds all variables, drives, commands, and the like that are used during the execution of a [PowerShell] object invocation. A runspace is always required when you want to execute PowerShell code, regardless of the mechanism used to execute that code, either API or script. A script user, however, typically isn’t aware that there is a runspace because it was created by the host (for example, the PowerShell console host or the PowerShell ISE) application at startup. And so far, we as API users, haven’t dealt with runspaces directly because the way we’ve been using the API allows the runtime to take care of the runspace requirement by creating a new one every time we call the API. This simplifies the API user’s experience but comes with constraints and significant execution overhead.

The major constraint coming from a new runspace on each execution is that you can’t incrementally build up state over a series of API calls. Conversely, a new runspace each time means that there is no cross-contamination between calls. This isolated execution is useful in its own right—for example, in creating uncontaminated test environments—and is something we’ll cover in more detail in the next section before moving on to the more general cases.

### 20.2.1 Existing Runspaces and Isolated Execution

In order for the Invoke() method on the [PowerShell] object to work, it needs an instance of the PowerShell runtime, namely a runspace. In all earlier examples, we didn’t worry about this because the runtime took care of it for us by creating a new runspace for every call to Invoke(). By creating a new runspace each time, we get isolated execution where side effects of one call can’t affect the operation of subsequent calls—at least as long as those side effects are restricted to runspace state. From an interactive user’s perspective, this includes isolation from the interactive PowerShell session.

Let’s walk through some examples to illustrate this behavior. You’ll do this by creating and assigning variables in the different environments. First, create a variable $x in the interactive session:

PS> $x = 123

then use the API to try to retrieve that value:

PS> [PowerShell]::Create().AddScript{$x}.Invoke()

Nothing is returned because the variable $x exists only in the interactive session, not in the new runspace created by the API. Now use the API to set the variable:

PS> [PowerShell]::Create().AddScript{$x=456}.Invoke()

and again, try to retrieve it:

PS> [PowerShell]::Create().AddScript{$x}.Invoke()

Nothing is returned because the assignment was made only in the transient runspace created by the API. Finally, you can verify that the original value of $x in the interactive session hasn’t changed:

PS> $x

123

#### Reusing the Current Runspace

Creating a runspace each time has some obvious limitations—sometimes you do want to preserve side effects across commands. Consider trying to preconfigure an isolated test environment. You’d execute a series of API calls to configure the environment before executing the test code with a separate API call. This scenario can’t work if you get a new environment every time you call the API.

The PowerShell API provides two mechanisms to accomplish durable state changes. The first allows you to say that the command should be run using the current runspace. The second involves your creating a durable environment in which to execute your commands (see [section 20.2.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-202-152.xhtml#ch20lev2sec6))

Let’s start with using the current runspace. If you’re in an interactive session, this would be the session’s runspace. This is done by passing an argument of type System.Management.Automation.RunspaceMode to the Create() method. This enum provides two values: CurrentRunspace and NewRunspace (the default). Let’s see an example using CurrentRunspace to change the value of $x you set up earlier:

PS> $x

123

PS> [PowerShell]::Create("CurrentRunspace").AddScript{$x=456}.Invoke()

PS> $x

456

This time, invoking the [PowerShell] object changes the value of $x in the session runspace. This is effectively equivalent to assigning the variable directly in the script. Given this, it’s not obvious why you’d want to use this version of the API—it certainly doesn’t provide any isolation. Where this can be useful is when you want to build up a pipeline dynamically and then execute it in the current runspace. [Listing 20.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-202-152.xhtml#ch20ex01) shows the implementation of a fancy file list, or fls, command built on top of Get-ChildItem. It allows you to sort files by newest first, selecting the first N files to list and setting the output to return only the full name of the item.

#### Listing 20.1: A fancy file list command

function fls

{

param (

[Parameter()]

[switch]

$New,

[Parameter()]

[int]

$First = -1,

[Parameter()]

[switch]

$NameOnly

)

$p = [PowerShell]::Create("CurrentRunspace"). 1

AddCommand("Get-ChildItem")

if ($New)

{

[void] $p.AddCommand("Sort-Object"). 2

AddParameter("Descending").

AddParameter("Property", "LastWriteTime")

}

if ($First -gt 0)

{

[void] $p.AddCommand("Select-Object"). 3

AddParameter("First", $First)

}

if ($NameOnly)

{

[void] $p.AddCommand("ForEach-Object"). 4

AddParameter("MemberName", "Fullname")

}

$p.Invoke()

if ($p.HadErrors) 5

{

$p.Streams.Errors

}

}

* 1 Create the base [PowerShell] object
* 2 If -New specified add sort command
* 3 Restrict output to $First N items
* 4 Change output and return only filename
* 5 Check to see if there were errors

This listing shows how commands can be built up incrementally and then be executed in the current runspace. Execution in the current runspace is necessary for the command to have access to the runspace’s current directory.

So far, we’ve been either dealing with runspaces that already exist or are automatically created on demand. Both of these cases limit what you can do with the runspace. In the next section, we’re going to look at how to explicitly create your own runspaces.

### 20.2.2 Creating Runspaces

Executing in your current session is useful, but a more interesting scenario would be to create a durable environment in which to execute your commands. This is core to the isolated test environment scenario we discussed earlier. Creating a durable execution environment is accomplished by explicitly creating a runspace and then using that runspace with the [PowerShell] API. Runspace creation is done using the System.Management.Automation.Runspaces.RunspaceFactory class, which has the type accelerator [runspacefactory]. This class provides methods that allow you to create a variety of runspace types. Let’s start with the simplest case.

Getting a usable runspace requires a couple of steps. First you create the runspace and then you open it:

PS> $rs = [runspacefactory]::CreateRunspace()

PS> $rs.Open()

Once the runspace is ready, you can create a [PowerShell] object and set the Runspace property on that object:

PS> $p = [PowerShell]::Create()

PS> $p.Runspace = $rs

By setting the runspace on the [PowerShell] object, you let the runtime know to use that runspace for execution rather than create a new one. With the runspace assigned, add a script to the [PowerShell] object in $p and invoke it:

PS> $p.AddScript{$x = 123}.Invoke()

The script that’s passed assigns a value to the variable $x in the associated runspace and so returns no value. Now you’re going to execute another command in that runspace. You could create a new [PowerShell] object and associate the runspace, but let’s look at an alternative way of doing this. Rather than creating a new [PowerShell] object each time, you can reuse the existing object by clearing the Commands property on the object. This removes all the previously added commands so you can start from scratch adding new commands:

PS> $p.Commands.Clear()

Now add a new script to return the value assigned to $x in the runspace you created. This will verify that its value is what you set it to in the first command:

PS> $p.AddScript{$x}.Invoke()

123

The output of the call to Invoke() confirms that the variable was set as intended.

By explicitly creating a separate runspace you now have two isolated execution environments for [PowerShell] commands. This is great for preventing cross-contamination, but another implication of two runspaces is that you should be able to do two things at once. You’ll see how this works in the next section.

### 20.2.3 Using Runspaces for Concurrency

Concurrent execution is important for real-world tasks where more than one thing happens at a time. PowerShell provides limited concurrent operations with Invoke -Expression fan-out (see [section 11.2.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-112-95.xhtml#ch11lev2sec8)) and background jobs (see [section 13.1.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-131-109.xhtml#ch13lev2sec2)). In this section, we’re going to look at how to perform concurrent operations using runspaces and the PowerShell API. The primary difference is in how you invoke the [PowerShell] object. In all earlier examples, you’ve been calling the Invoke() method. This is a synchronous method that starts execution and then waits for it to complete, returning the result of the execution. This prevents the caller’s runspace thread (the foreground thread) from doing anything until the second runspace (the background thread of execution) has completed. In order to execute operations concurrently, you need a way to begin an asynchronous thread of execution. With the [PowerShell] object, this is done using the BeginInvoke() method.

Whereas the Invoke() method blocks until the execution completes and returns the result of that execution, the BeginInvoke() method immediately returns an object of type IASyncResult. This IASyncResult object provides a way for you to interoperate with the asynchronous operation you started. The most basic signature for BeginInvoke() is System.IAsyncResult BeginInvoke(). Let’s look at an example to see what the IASyncResult object tells us about the background execution. Create a [PowerShell] object with a single command and call BeginInvoke():

PS> $ia = [PowerShell]::Create().AddCommand("Get-Date").BeginInvoke()

#### Note

In this example, you’re being a bit lazy and letting the runtime create the background runspace for you. Though simple, this doesn’t allow for runspace reuse and so is not generally recommended as a best practice.

In the example, you’re capturing the IASyncResult from the execution into the variable $ia so you can work with it later on. Let’s display the object formatted as a list:

PS> $ia | Format-List

CompletedSynchronously : False

IsCompleted : True

AsyncState :

AsyncWaitHandle : System.Threading.ManualResetEvent

The most important property for your immediate purposes is the IsCompleted property. This lets you know that the background execution has completed. In this simple example, the IsCompleted property is true immediately because the background execution was short. Now try running a command that takes longer. The Start-Sleep command is a good choice because you can specify fairly precisely how long you want the command to run:

PS> $ia = [PowerShell]::Create().AddCommand("Start-Sleep"). `

AddParameter("Seconds",5).BeginInvoke()

PS> $ia.IsCompleted

False

This time when you examine IsCompleted, you can see that the execution has not completed. Checking again in a few seconds, you’ll see that it has completed:

PS> Start-Sleep -Seconds 5 ; $ia.IsCompleted

True

With commands other than Start-Sleep, the amount of time the command will take to complete is harder to predict. Clearly there has to be a better solution to waiting for completion than continuously checking (polling) the IsCompleted property. This is where the EndInvoke() method on the [PowerShell] object comes in. You pass the IASyncResult object returned from BeginInvoke() to EndInvoke(), and the foreground thread of execution will block until the background execution has completed. To do this, you need to store the [PowerShell] object in a variable in order to call EndInvoke() on that object. This looks like the following:

PS> $p = [PowerShell]::Create().AddCommand("Start-Sleep"). `

AddParameter("Seconds",5)

PS> $ia = $p.BeginInvoke(); $p.EndInvoke($ia)

PS> $ia.IsCompleted

True

If you check the value of IsCompleted after calling EndInvoke(), it will always be true. In effect, the BeginInvoke()/EndInvoke() pair are equivalent to the synchronous Invoke() except both threads run in parallel until EndInvoke() is called. Let’s look at an example ([listing 20.2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-202-152.xhtml#ch20ex02)) where you can see that both the foreground and background execution threads are running concurrently.

#### Note

This example calls the Windows [console] API to print messages on the screen so you can see that they’re both operating. The implication is that if you run it in the ISE, it won’t work as expected.

#### Listing 20.2: Concurrent execution example

$r = [runspacefactory]::CreateRunspace() 1

$r.Open()

$p = [PowerShell]::Create().AddScript{ 2

foreach ($i in 1..4) {

[console]::WriteLine(">>> BACKGROUND $i")

Start-Sleep 1

}

[console]::WriteLine("Background is done")

}

$p.Runspace = $r

$a = $p.BeginInvoke() 3

foreach ($i in 1..3) {

[console]::WriteLine("foreground $i <<<")

Start-Sleep 1

}

[console]::WriteLine("Foreground is done")

$p.EndInvoke($a) 4

"Called EndInvoke."

* 1 Create a background runspace
* 2 Define the background task
* 3 Start the background task
* 4 Wait for the background task

Running this script from the PowerShell console host produces the following output:

foreground 1 <<<

>>> BACKGROUND 1

foreground 2 <<<

>>> BACKGROUND 2

foreground 3 <<<

>>> BACKGROUND 3

Foreground is done

>>> BACKGROUND 4

Background is done

Called EndInvoke.

The messages from the foreground and background runspaces are interleaved. Because the background task does four iterations and the foreground task does only three, the foreground task completes first and then waits for the background task by calling EndInvoke().

This is a trivial example. A more realistic example would be to perform several related, long-running operations concurrently, such as large file copies, formatting a disk, or creating virtual machines. In these scenarios, there may be a fairly large number of operations that could be performed in parallel. Manually creating and managing a large number of runspaces for these scenarios could be quite complex. It would be nice if PowerShell took care of all this bookkeeping is some way. That’s exactly what runspace pools are all about. We’ll look at those objects in detail in the next section.

## 20.3 Runspace Pools

In all examples so far, you’ve been creating individual runspaces for each of the tasks you’re performing. This results in numerous runspaces being created. Explicit reuse will reduce the number of runspaces that are created but there may still be a lot of work tracking all of them.

PowerShell provides a mechanism called runspace pools to take care of this bookkeeping automatically. A single runspace pool is made up of a number of individual runspaces. The runspace pool API allows you to set a number of constraints on the pool, allowing for automatic management of the amount of resources consumed. This is called throttling. For example, a runspace pool will allow you to limit (or throttle) the number of concurrent operations without having to explicitly code what’s going on. You can start as many tasks as you need without worrying about running out of resources on the host machine. The runspace pool does this by limiting the pool of runspaces from a minimum to a maximum number of runspaces. Here’s an example showing the creation of a runspace pool with a minimum of one and a maximum of three runspaces:

PS> $pool = [runspacefactory]::CreateRunspacePool(1, 3)

PS> $pool.Open()

PS> $pool.GetAvailableRunspaces()

3

When the pool is opened, it will have one runspace open and available. Now let’s start a command running and see how the runspace count changes:

PS> $p1 = [PowerShell]::Create().AddCommand("Start-Sleep").AddArgument(30)

PS> $p1.RunspacePool = $pool

PS> $ia1 = $p1.BeginInvoke()

PS> $pool.GetAvailableRunspaces()

2

Add two more tasks:

PS> $p2 = [PowerShell]::Create().AddCommand("Start-Sleep").AddArgument(30)

PS> $p2.RunspacePool = $pool

PS> $ia2 = $p2.BeginInvoke()

PS> $p3 = [PowerShell]::Create().AddCommand("Start-Sleep").AddArgument(30)

PS> $p3.RunspacePool = $pool

PS> $ia3 = $p3.BeginInvoke()

PS> $pool.GetAvailableRunspaces()

0

The number of available runspaces drops to zero. But you can still add tasks to the pool even though there are zero available runspaces at that time:

PS> $p4 = [PowerShell]::Create().AddCommand("Start-Sleep").AddArgument(30)

PS> $p4.RunspacePool = $pool

PS> $ia4 = $p4.BeginInvoke()

PS> $pool.GetAvailableRunspaces()

0

When there are no available runspaces in the pool, new tasks are placed in a queue of tasks waiting to be executed. When a running task completes and its runspace becomes available, the next task in the queue is removed and invoked on the newly available runspace. The pool will continue to execute the maximum concurrent tasks allowed until the task queue is empty.

And now back to reality. Runspace pools are efficient mechanisms for handling concurrent operations, but you still need to deal with errors, which means you need to keep track of all the PowerShell objects you’re creating. This listing shows how to do this.

**Listing 20.3: Foreach in parallel**

$pool = [runspacefactory]::CreateRunspacePool(1, 3) 1

$pool.Open()

$tasks = foreach ($i in 1 .. 10) 2

{

$p = [PowerShell]::Create()

$p.RunspacePool = $pool

$p = $p.AddScript{ 3

param ($iteration)

foreach ($i in 1..5)

{

[console]::WriteLine("\*" \* ($iteration \* 2)) 4

Start-Sleep -Milliseconds 200

}

if ($iteration -eq 3) 5

{

Write-Error "ITERATION ERROR"

}

}.AddArgument($i)

$ia = $p.BeginInvoke()

@{p=$p; ia=$ia; iteration=$i} 6

}

foreach ($t in $tasks)

{

$t.p.EndInvoke($t.ia)

if ($t.p.HadErrors) 7

{

Write-Error "Task iteration $($t.iteration) had errors"

$t.p.Streams.Errors

}

$t.p.Dispose()

}

* 1 Limit to three concurrent tasks
* 2 Capture the information for each task
* 3 Set the code for the task
* 4 Each task writes iteration\*2 stars to the console
* 5 The third iteration will write an error
* 6 Capture [PowerShell], await object and iteration number
* 7 Check each iteration for errors

This example starts 10 tasks. Each task writes a line of stars to the console five times, where the number of stars written corresponds to the index of the task in the list of tasks being executed. The number of stars to write is passed as an argument to the task scriptblock, which is the same for each task.

**Note**

In parallel processing terminology, this is called single instruction, multiple data (SIMD). Other variations would be passing both a unique scriptblock and a unique piece of data to each task, called multiple instruction, multiple data (MIMD), and finally passing multiple scripts but always using the same piece of data, or multiple instruction, single data (MISD).

To make sure at least one error is produced, there is special logic so that when the task index is 3, an error message is written.

To track all this information, as each task is started its [PowerShell] object, IASync-Handle, and task index are put into a hashtable that is then written to the output stream of the foreach statement. This output is captured in a variable, $tasks, and once all the tasks have started, a second foreach loop takes the complete list of tasks, waits for each task to complete, and then checks to see if any errors occurred during the execution of that task. Running this script will produce output similar to the following:

PS > .\foreachparallel.ps1

\*\*

\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*

\*\*

\*\*

\*\*\*\*\*\*

\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*

\*\*

\*\*\*\*

\*\*

\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

C:\Users\brucepay\documents\foreachparallel.ps1 : Task iteration 3 had errors

At line:1 char:1

+ .\foreachparallel.ps1

+ ~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : NotSpecified: (:) [Write-Error], WriteErrorException

+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,foreachparallel.ps1

\*\*\*\*\*\*\*\*\*\*

<output truncated for brevity>

Note that the error message was printed before that last line of stars. The tasks are waited for in the order in which they were started, and so an individual task will complete before all the tasks have completed. If you wanted to defer error checking until all the tasks have completed, you would process the contents of $tasks twice—the first time to make sure all the tasks have completed, and then a second time to make sure all the errors have been accounted for.

To apply concurrent techniques successfully, you need to be careful about tracking the task objects and the associated errors. Taking advantage of the natural flow of the PowerShell pipeline makes this easy. The alternative—explicitly creating a collection and adding each task to the collection—makes the resulting code significantly more complex. For concurrent techniques to be successful, you need to take a disciplined approach that minimizes complexity and, in particular, approaches error handling in a structured way.

All the techniques we’ve looked at so far have used in-memory runspaces. In-memory runspaces have lower overhead than out-of-process executions like the jobs created with Start-Job, but there are cases where you want to have that extra layer of isolation. With in-memory runspaces, if one of the tasks causes the process to terminate, all the other tasks will also be terminated. With process isolation, a task that crashes the process crashes only its host process. In the next section, we’ll look at using process isolation with runspaces.

## 20.4 Out-of-Process Runspaces

Let’s revisit the idea of isolation with runspaces. In [section 20.2.1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-202-152.xhtml#ch20lev2sec5), we talked about how a runspace isolates all the PowerShell-specific data structures. This means that any process-wide pieces of data are still shared by all the runspaces. It also means that a catastrophic error—one that will cause the host process to exit—will also terminate all the runspaces in the process.

To provide an even greater layer of isolation, you can create an out-of-process runspace. An out-of-process runspace is created in a new process that is a child of the calling process and uses the PowerShell remoting protocol to communicate between the two processes. This is somewhat similar to the way Start-Job jobs work, but an out-of-process runspace lifecycle is different. With Start-Job, a new process is created when the job begins. The process exists for the duration of the job and then terminates when the job has completed. With out-of-process runspaces, the process is started when the runspace is opened and isn’t terminated until the runspace is closed. This means you can run many tasks in the same process by reusing the runspace. It also means out-of-process runspaces are quite a bit more capable than background jobs but aren’t significantly more complex than in-memory runspaces.

Creating an out-of-process runspace is straightforward: call the CreateOutOfProcessRunspace() method to create the runspace and then open it like any other runspace:

PS> $ooprs = [runspacefactory]::CreateOutOfProcessRunspace($null)

PS> $ooprs.Open()

**Note**

The first argument to CreateOutOfProcessRunspace() is a pointer to an optional custom type table to use when communicating with that runspace. If this argument is null, the default PowerShell type table is used. PowerShell’s default type files are loaded into the runspace just as if you’d started an instance of PowerShell. This default is fine in the majority of cases, but sometimes you may want specific control over how objects are serialized when passed between the processes. Custom type files allow this custom serialization information to be used by the remoting protocol. For information on how to create a type table instance, see <http://mng.bz/46M0>.

Once the runspace is open, you can send commands to it, like any other runspace. As an example, let’s verify that the runspace is hosted in a separate process. You’ll use the process identifier (PID), which is unique to each process, to distinguish parent from child. You’ll be sending a command to the runspace that will return the process’s ID using the $PID variable. First, create the [PowerShell] object for the command:

PS> $p = [PowerShell]::Create().AddScript{"child PID is $PID"}

Then set the runspace on the [PowerShell] object and invoke the command:

PS> $p.Runspace = $ooprs

PS> $p.Invoke()

child PID is 196

Now that you have the runspace PID, get the interactive host process id:

PS> "Local pid is $pid"

Local pid is 8368

On examination of the two PIDs, you can verify that the runspace is running in a different process.

So far, we’ve covered in-process and out-of-process runspaces. The last type of runspace we’re going to cover is the remote runspace.

## 20.5 Remote Runspaces

So far, all our runspace work has been done on the local computer. Now we’re going to look at working with runspaces running on remote computers. But first a quick word about sessions and runspaces.

### 20.5.1 Sessions and Runspaces

In [chapter 11](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-11-93.xhtml#ch11), when talking about remoting, we always talked about remote connections requiring sessions but never about remote runspaces. In the early design of PowerShell remoting, there were no PSSession objects, only runspaces. But when the team did usability studies, people were much more comfortable with the notion of a remote session because the term was already commonly used. The team introduced the PSSession object to be the script user-facing term, with the Runspace being relegated to advanced scenarios.

#### Note

Now that PowerShell is open source, you can see this session-versus-runspace legacy in the source itself. The file that contains the code for the New-PSSession command is still named newrunspacecommand.cs.

Fortunately, obtaining the underlying runspace from a session is quite simple because it’s available as a property on the PSSession object:

PS> $s = New-PSSession localhost

PS> $s.Runspace

Id Name ComputerName Type State Availability

-- ---- ------------ ---- ----- ------------

35 Runspace35 localhost Remote Opened Available

### 20.5.2 Creating Remote Runspaces

Creating a remote runspace follows the same pattern you’ve used all along: You call the [runspacefactory]::CreateRunspace() method to the runspace. The difference between creating a remote versus a local runspace is that you must supply information about how to connect to the remote computer. This is done using the System.Management.Automation.Runspaces.WSManConnectionInfo class. Here’s an example. Create the WSManConnectionInfo object by calling the constructor on it:

PS> $ci =

[System.Management.Automation.Runspaces.WSManConnectionInfo]::new()

Let’s look at a subset of the information contained in the connection information object:

PS> $ci | Format-List scheme,computerName,port,appname

Scheme : http

ComputerName : localhost

Port : 80

AppName : /wsman

This shows that you want to connect to the WS-MAN application on computer localhost using port 80 and HTTP for the base transport. You can now use this object to create a remote runspace by passing it to the CreateRunspace() method:

PS> $rrs = [runspacefactory]::CreateRunspace($ci)

PS> $rrs.GetType().FullName

System.Management.Automation.RemoteRunspace

Checking the type, you can see that the method has returned a RemoteRunspace instead of a regular Runspace. As always, before you can use a runspace to execute any code, it needs to be opened:

PS> $rrs.open()

Now create a [PowerShell] object and set its Runspace property to the remote runspace you created:

PS> $p = [PowerShell]::Create()

PS> $p.Runspace = $rrs

Add a scriptblock to the [PowerShell] object to print out the computer’s hostname and the PID of the process hosting

PS> $p = $p.AddScript{

"I am on host $(hostname)"

"My PID is $pid"

}

and invoke it:

PS> $p.Invoke()

I am on host brucepaybook

My PID is 17356

The output shows that the remote PID is 17356 whereas the local PID is 1132

PS> "Local PID is $PID"

Local PID is 1132

confirming, as we did in the out-of-process runspace case, that different processes are being used to host the runspaces.

#### Note

You may now be wondering how this scenario differs from the out-of-process case. In that case, the two processes are communicating directly over anonymous pipes. In the remoting case, the local session is communicating to the WS-MAN application, which creates the remote process and manages communication between the local and remote processes. Because there is an intermediary (WS-MAN), remote runspaces can support more features, including disconnected runspaces. In the out-of-process case, if the parent process terminates, then the child process is also terminated. In the remote runspace case, the local session can terminate, but the remote can remain active because the WS-MAN service manages the lifecycle of the remote session.

This concludes our rather brief discussion of remote runspaces. For more detailed information, consult the MSDN pages for the APIs, in particular the [WSManConnectionInfo] class. See <http://mng.bz/BCPq>. The final section of this chapter looks at some basic hygiene principles with managing runspaces.

## 20.6 Managing Runspaces

One last topic before we go: runspace management. We’ve been proceeding along, opening runspaces as needed. What we haven’t always been doing is cleaning up the runspaces that we’ve been creating. Once a runspace is created and opened, it will remain in the current process until it’s explicitly closed. You can find out how many runspaces you have going with the Get-Runspace command. Let’s examine this further. Starting from a new session, create and open two runspaces:

PS> $r1 = [runspacefactory]::CreateRunspace()

PS> $r1.Open()

PS> $r2 = [runspacefactory]::CreateRunspace()

PS> $r2.Open()

Now use Get-Runspace to list all the runspaces in the session:

PS > Get-RunSpace

Id Name ComputerName Type State Availability

-- ---- ------------ ---- ----- ------------

1 Runspace1 localhost Local Opened Busy

2 Runspace2 localhost Local Opened Available

3 Runspace3 localhost Local Opened Available

Notice that there are three runspaces in this session. That’s because of the default runspace (Id 1) that handles the interactive commands passed to the session. Each runspace has a set of properties associated with it. The State property is fairly obvious; it’s the current state of the runspace. In the formatted output this shows up as State, but note that state is a computed property in the format information that’s equivalent to

PS> (Get-Runspace)[0].RunspaceStateInfo.State

Opened

The other property to be aware of is Availability. This property indicates whether there’s a pipeline currently running in the runspace.

**Note**

Availability is also an alias used in formatting so things will fit on the screen. The property name is RunspaceAvailability.

Note that the default runspace will always be busy when you run a command because that runspace is used to run the command you typed. To close a runspace you need to call the close method on that runspace. The trick is not to close the default runspace, because that will end your session. You can do this by filtering on the RunspaceAvailability property as follows:

PS> Get-Runspace |

where { $\_.RunspaceAvailability -eq "Available" } |

foreach Close

Now when you rerun the Get-Runspace command, all the non-default runspaces are closed:

PS> Get-RunSpace

Id Name ComputerName Type State Availability

-- ---- ------------ ---- ----- ------------

1 Runspace1 localhost Local Opened Busy

2 Runspace2 localhost Local Closed None

3 Runspace3 localhost Local Closed None

Runspace management isn’t a major concern when working interactively, but if you have a long-running script that continually opens new runspaces without closing old ones, eventually you’ll consume all the available resources. A much better strategy when using runspaces in a script is to create the necessary number of runspaces and then reuse them as needed. If you don’t know how many runspaces you’ll need, then the best solution is runspace pools, as discussed in [section 20.3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-203-153.xhtml#ch20lev1sec3).

## 20.7 Summary

* The [PowerShell] API is a programmatic mechanism for creating and invoking PowerShell commands.
* This API can be used to build up multistage pipelines, both in fluent expressions and incrementally.
* This API also provides mechanisms for handling errors and exceptions in the pipelines you’re building.
* Scriptblocks and statements can be added to your [PowerShell] expressions.
* Runspaces are created using the [runspacefactory] API.
* Runspaces can be used to create isolated execution environments and to perform concurrent operations.
* Runspace pools make it easier to deal with concurrent scenarios where there are many threads of execution.
* Out-of-process runspaces provide an additional level of isolation for tasks.
* Remote runspaces are used with the [PowerShell] API to execute commands on remote machines.

## Appendix. PowerShell 6.0 for Windows, Linux, and macOS

### Overview

The PowerShell community was stunned (not too dramatic a word) in August 2016 when Microsoft announced that the core of PowerShell was going to be open source. Not only that, but PowerShell would now be available on Linux and Apple operating systems as well as Windows. Jeffrey Snover had been hinting in presentations at the PowerShell Summit (<https://powershell.org/summit/>) and other venues for a few years that he’d like to see PowerShell as an open source project, but it wasn’t expected to happen so soon.

**Note**

The open source project includes the core PowerShell engine. Many of the non-core modules will also be ported. Many existing modules should work with PowerShell v6 due to the use of .NET standard 2.0.

In this appendix, we’ll give you an overview of the open source project, explain the differences between PowerShell Core and the PowerShell you find on your Windows machine, and demonstrate the differences between running PowerShell on Linux/macOS compared to the experience you have on Windows.

## The PowerShell Open Source Project

The open source PowerShell project is hosted on Github at <https://github.com/PowerShell/PowerShell> and is open for anyone to join and contribute. PowerShell 6.0 includes versions for Windows, most major Linux distributions, macOS, and Docker. The list of supported platforms is evolving. You can see a complete list of supported platforms at the GitHub site. You can install PowerShell 6.0 side by side on Windows with an existing instance of PowerShell.

#### PowerShell 6.0 code status

At the time of writing PowerShell v6 is in development. It should be considered test code until it’s formally released. Check the project website for the current state of the project before using.

### Terminology

PowerShell v6 has introduced new terminology. [Table 1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-159.xhtml#app01table01) summarizes the terminology and explains the various editions of PowerShell.

| **Table 1: PowerShell terminology** | |
| --- | --- |
| **Term** | **Meaning** |
| Windows PowerShell | This is the edition of PowerShell that ships with Windows (or in a WMF download). It’s built in, and requires, the full .NET CLR. Windows PowerShell is only available on the Windows platform. $PSVersionTable.PSEdition is set to Desktop. |
| PowerShell Core (PSCore) | This edition of PowerShell is built on the .NET Core CLR (see next section). PowerShell Core will be available on all supported platforms. $PSVersionTable.PSEdition is set to Core. |
| PowerShell on ? | PowerShell Core built for a specific platform, for instance: PowerShell on Linux or more specifically PowerShell on Centos 7. |
| PowerShell | A generic term that covers any and all editions. PowerShell can be used to refer to the language, framework, default cmdlets, and so on. |

We’ve mentioned .NET Core a few times. It’s time to explain what it is and how it’s different from the .NET you’ve seen and used on Windows.

### .NET Core

Throughout this book we’ve said that PowerShell is based on the .NET framework. This remains true for all editions of PowerShell. However, not all editions of PowerShell use the full .NET framework. The use of .NET is as follows:

* Windows PowerShell uses the full .NET CLR.
* PowerShell Core (PowerShell on Linux and so on) uses .NET Core.

The full .NET framework is described on Microsoft’s MSDN site at <http://mng.bz/PTPZ>. All the examples in [chapters 1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-1-8.xhtml#ch01)–[20](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-20-150.xhtml#ch20) are based on Windows and therefore on the full .NET CLR.

PowerShell Core is based on .NET Core, which is a cross-platform implementation of the Windows .NET framework.

#### Note

.NET Core is technically a subset of the full .NET CLR, but the .NET Standard 2.0 release of .NET Core has dramatically reduced the delta between the full CLR and Core. There will also be an ability to load full CLR assemblies into Core processes.

.NET Core is available for Windows, Linux, and macOS. This means that scripts written on Windows accessing the full .NET framework may not run on a non-Windows platform due to functionality not being present in .NET Core. You can view the .NET Core DLLs supplied with PowerShell Core and get a good idea of the .NET functionality available:

PS /home/richard> Get-ChildItem -path $pshome/\*.dll

References for .NET Core can be found at <http://www.dotnetfoundation.org/netcore> and <http://www.microsoft.com/net/core/platform>.

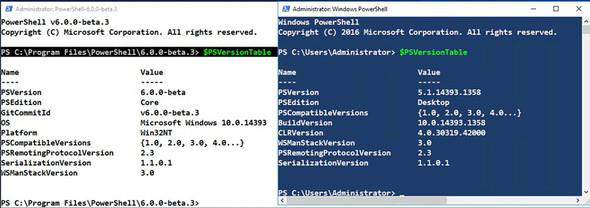
PowerShell Core doesn’t port all the PowerShell functionality you’re used to, so there will be some issues (covered when we look at PowerShell on Linux). For an existing PowerShell user, probably the best place to start is by installing PowerShell Core onto an existing Windows system.

### Installing on Windows

Installation on Windows uses a standard Windows .msi. Download the latest release for your version of Windows from <https://github.com/PowerShell/PowerShell/releases>. It’ll be named something like PowerShell-6.0.0-beta.3-win10-win2016-x64.msi. After unblocking the file, double-click the .msi and follow the instructions. PowerShell 6.0 installs into C:\Program Files\PowerShell by default. An entry is also created on the Start menu.

You can run PowerShell 6.0 side by side with Windows PowerShell (v5.1 in this case), as shown in [figure 1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-159.xhtml#app01fig01). Notice the differences between the two sets of output, especially the value of PsEdition.

Figure 1: PowerShell 6.0 on the left and PowerShell 5.1 on the right. The background and text colors have been reversed from the default in the PowerShell 6.0 console for clarity



You get a limited number of modules, and therefore cmdlets, available in PowerShell v6 on Windows:

PS> Get-Module -ListAvailable | select name

Name

----

CimCmdlets

Microsoft.PowerShell.Archive

Microsoft.PowerShell.Diagnostics

Microsoft.PowerShell.Host

Microsoft.PowerShell.LocalAccounts

Microsoft.PowerShell.Management

Microsoft.PowerShell.Security

Microsoft.PowerShell.Utility

Microsoft.WSMan.Management

PackageManagement

Pester

PowerShellGet

PSDesiredStateConfiguration

PSDiagnostics

PSReadLine

You can create a remote session to a machine running a copy of Windows PowerShell (in this case, PowerShell 5.1):

PS> $s = New-PSSession -ComputerName W16DSC01

PS> Invoke-Command -Session $s -ScriptBlock {Get-Process l\*} |

Format-Table -AutoSize

NPM(K) PM(M) WS(M) CPU(s) Id SI ProcessName PSComputerName

------ ----- ----- ------ -- -- ----------- --------------

23 10.09 32.87 0.59 740 1 LogonUI W16DSC01

30 5.77 8.96 1.22 524 0 lsass W16DSC01

Cmdlets that aren’t part of PowerShell Core can be accessed on the remote machine:

PS> Invoke-Command -Session $s `

-ScriptBlock {Get-WmiObject -Class Win32\_OperatingSystem}

SystemDirectory : C:\Windows\system32

Organization :

BuildNumber : 14393

RegisteredUser : Windows User

SerialNumber : 00376-30816-46802-AA030

Version : 10.0.14393

PSComputerName : W16DSC01

Within the limitations described earlier, PowerShell on Windows is very similar to the PowerShell you’ve seen throughout this book. What about PowerShell on Linux and macOS?

## PowerShell on Linux and macOS

The introduction of PowerShell for Linux/macOS is a big step forward for managing heterogeneous environments. Windows administrators can now manage these systems using the same tool—PowerShell—they’re used to using on their Windows systems. In this section, we cover installing PowerShell on Linux and the differences between PowerShell on Windows and PowerShell on Linux.

PowerShell is built on the assumption that you’ll be administering your systems remotely. We examine remoting from Windows to Linux and Linux to Windows systems. DSC brings a huge change in the way you manage the configuration of your servers. You can manage Linux systems as well as Windows with DSC.

There are some differences, and issues, between Windows PowerShell and PowerShell on Linux/macOS.

### Known Issues

As you would expect, there are a number of issues with porting a 10-year-old .NET-based application—PowerShell—to non-Windows platforms. The PowerShell project team maintains a list of known issues as part of the project documentation at <http://mng.bz/8j3L>.

Some of these issues are flagged to be addressed during the development of PowerShell v6. Other issues are differences that are inherent to the various platforms to which PowerShell is being ported, and users will need to be aware of the issues and manage them.

Issues that may cause current Windows-based PowerShell users problems on non-Windows platforms or that may cause issues for non-Windows users learning PowerShell include the following:

* **Case sensitivity—** PowerShell, like Windows, is case-insensitive. Linux and macOS are case-sensitive, so the correct case must be used for filenames, paths, and environment variables. Running scripts, loading modules, and filename tab completion all depend on the correct case being used. Cmdlet names are case-insensitive!
* **File path delimiters—** Windows can use \ or /, but on non-Windows you must use /.
* **File extensions—** PowerShell uses file extensions—for instance, .ps1 for scripts and .psm1 for modules. Non-Windows platforms don’t usually use file extensions. You need to use the correct extension for PowerShell to correctly interpret the file type.
* **Command aliases—** A number of aliases—ls, cp, mv, rm, cat, man, mount, ps—have been removed from the Linux and macOS implementations, as they hide the platform-native commands. These aliases are still present in PowerShell for Windows.
* **JEA—** JEA support is not available on Linux or macOS and is not in scope for PowerShell v6.
* **Sudo—** PowerShell doesn’t support sudo directly. You need to start a new instance of PowerShell using sudo.
* **Missing cmdlets—** A number of cmdlets don’t work properly or aren’t available on Linux and macOS, including \*-Service, \*-Acl, \*-AuthenticodeSignatue, Wait-Process, \*-PSSessionConfiguration, \*-Event, Set-ExceutionPolicy, New-PSSession, New-PSSessionOption, New-PSTransportOption, and \*-Job. Some of these issues will be resolved in future releases.

One other thing to be aware of is that none of the PowerShell Core implementations includes the PowerShell ISE. If you need an editor for use with PowerShell Core, especially on non-Windows machines, we recommend using Visual Studio Code (VSC). VSC is a free download from <https://code.visualstudio.com/> with versions available for Windows, various Linux distributions, and macOS. PowerShell and many other programming languages are supported through plugins that can be installed from within Visual Studio Code.

Before you can do anything, though, you need to get PowerShell onto your Linux system.

### Installation

PowerShell on Linux is available on a large number of Linux distributions, but in this appendix we’ll just be looking at PowerShell on Centos 7.

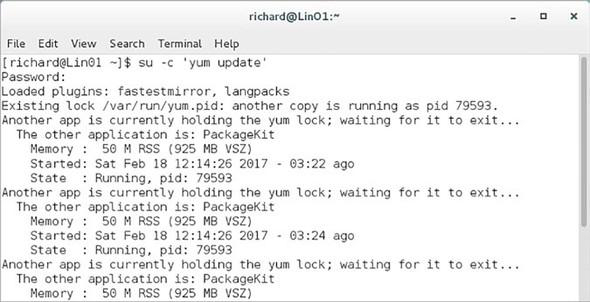
#### Note

We’ll be assuming that you have a working Linux system and have sufficient Linux skills to follow this discussion.

You can find instructions for installing PowerShell on CentOS 7 at <http://mng.bz/7fa8>. Instructions for other Linux types and macOS are also available. Follow the download instructions and install the PowerShell package.

You may find that you see a message about a yum lock, as shown in [figure 2](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-160.xhtml#app01fig02).

Figure 2: Yum error message due to lock held by PackageKit



If you get such a message, follow the instructions to disable packagekit (which manages updates) that you’ll find at <http://mng.bz/5co7>. The PowerShell package can then be installed.

The PowerShell project has started releasing the install packages to the appropriate repositories so that you can use Linux’s built-in package management systems to install and update PowerShell. First, you need to enter super user mode and register the Microsoft repository:

sudo su

curl https://packages.microsoft.com/config/rhel/7/prod.repo > /etc/yum.repos.d/microsoft.repo

exit

You can then install PowerShell:

sudo yum install -y powershell

Starting PowerShell is a simple call to the application:

powershell

The advantage of this approach is that when new releases of PowerShell 6.0 are made available, you can easily update your installation:

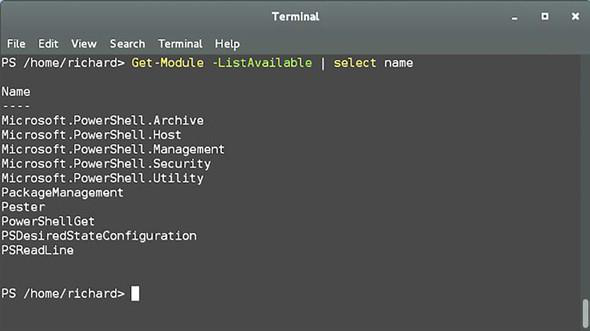
sudo yum update powershell

Now that you’ve installed PowerShell for Linux, how do you use it?

### Using PowerShell v6 on Linux

Using the PowerShell core language is essentially identical to the examples we’ve shown in the rest of the book. One obvious difference is the modules that are available. You saw the modules available for PowerShell on Windows earlier. [Figure 3](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-160.xhtml#app01fig03) shows the modules available to a new PowerShell on Linux installation.

Figure 3: A list of default modules for PowerShell on Linux



A number of modules that are available in PowerShell for Windows aren’t available in PowerShell for Linux:

* CimCmdlets
* Microsoft.PowerShell.Diagnostics
* Microsoft.PowerShell.LocalAccounts
* Microsoft.WSMan.Management
* PSDiagnostics

These modules contain functionality that is directly related to the Windows platform and so can’t be ported to Linux and other platforms. Linux/macOS versions of these modules may become available in the future. There are some possible issues with other modules. Script modules will load but may not work properly if they make Windows-centric assumptions about the file system or access Windows-specific functionality. Binary modules won’t load if they depend on functionality that isn’t present in .NET Standard 2.0.

A number of the Windows PowerShell providers are also not available on Linux/macOS:

* Registry
* WSMan
* Certificate

PowerShell v6 being available on a number of platforms means you can write scripts that are portable across platforms. Your script needs to know which platform it’s running on to avoid errors due to missing functionality. The $PSVersionTable contains detailed operating system information. On Windows, you’ll see this:

PS> $PSVersionTable

Name Value

---- -----

PSVersion 6.0.0-beta

PSEdition Core

GitCommitId v6.0.0-beta.3

OS Microsoft Windows 10.0.14393

Platform Win32NT

PSCompatibleVersions {1.0, 2.0, 3.0, 4.0...}

PSRemotingProtocolVersion 2.3

SerializationVersion 1.1.0.1

WSManStackVersion 3.0

On Linux you’ll see this:

PS /home/richard> $PSVersionTable

Name Value

---- -----

PSVersion 6.0.0-beta

PSEdition Core

GitCommitId v6.0.0-beta.3

OS Linux 3.10.0-514.6.1.el7.x86\_64

#1 SMP Wed Jan 18 13:06:36 UTC 2017

Platform Unix

PSCompatibleVersions {1.0, 2.0, 3.0, 4.0...}

PSRemotingProtocolVersion 2.3

SerializationVersion 1.1.0.1

WSManStackVersion 3.0

The Platform field or the OS field (if you require more detailed tests) can be used to determine the operating system the code is running on.

PowerShell v6 includes variables, shown in [listing 1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-160.xhtml#app01ex01), to help with this:

* $IsCoreCLR
* $IsLinux
* $IsOSX
* $IsWindows

#### Listing 1: Cross-platform scripting

$dt = @{

'3' = 'Fixed'

'5' = 'CD-Rom'

}

$hlth = @{

'0' = 'Healthy'

'1' = 'Scan Needed'

'3' = 'Full Repair Needed'

}

if ($IsCoreCLR) {

if ($IsLinux){

df -T

}

elseif ($IsWindows) {

Get-CimInstance -Namespace 'ROOT/Microsoft/Windows/Storage' `

-ClassName MSFT\_Volume |

select DriveLetter, FileSystemLabel, FileSystem,

@{N='DriveType'; E={$dt["$($\_.DriveType)"]}},

@{N='HealthStatus'; E={$hlth["$($\_.HealthStatus)"]}},

@{N='SizeRemaining(GB)'; E={[math]::Round($\_.SizeRemaining / 1GB, 2)}},

@{N='Size(GB)'; E={[math]::Round($\_.Size / 1GB, 2)}}

}

}

else {

Get-Volume

}

The script starts by defining two hash tables, $dt and $hlth, that will be used to decode the values returned from a CIM class. The variable $IsCoreCLR is used to determine whether the script is running on an instance of PowerShell Core or Windows PowerShell.

If the script is running on PowerShell Core, the next test determines if it’s running on Linux or Windows. The Linux command df -T is used to return disk information if the script is running on Linux. Get-CimInstance with a call to the MSFT\_Volume class is used for a Windows machine running PowerShell Core. The hash tables are used to supply readable values for the drive type and health status. The disk size information is converted to GB with the result converted to two decimal places.

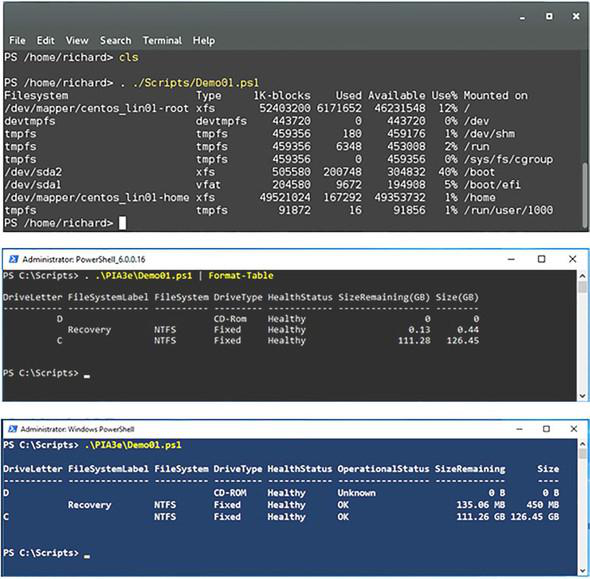
If the script isn’t running on PowerShell Core, it assumes Windows PowerShell and uses the Get-Volume cmdlet. Get-Volume also uses the MSFT\_Volume CIM class.

#### Note

The Get-Volume cmdlet and the MSFT\_Volume CIM class are only available on Windows 8/2012 and later. If you're using Windows 7, you can modify the script to use Win32\_Volume instead.

The results of running the script are shown in [figure 4](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-160.xhtml#app01fig04).

Figure 4: The results of running [listing 1](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-160.xhtml#app01ex01) on PowerShell on Linux (top), PowerShell on Windows (middle), and Windows PowerShell (bottom)



So far, you’ve seen PowerShell running directly on Linux. What about remoting between instances of PowerShell running on Windows and Linux?

## PowerShell Remoting and Linux

You saw in [chapter 11](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-11-93.xhtml#ch11) how PowerShell remoting works using WS-MAN as its transport mechanism. PowerShell for Linux doesn’t include a WS-MAN provider. Traditionally, Linux has used SSH (Secure Shell) for remote access. PowerShell for Linux performs remote access using SSH.

**Note**

The plan is that eventually there will be a single mechanism for PowerShell remoting regardless of the client and target. Until that time, you need to use SSH if Linux machines are involved and WS-MAN if only Windows machines are involved.

If you want to perform PowerShell remoting between Windows machines and Linux machines, you need to have SSH installed at both ends. PowerShell on Windows uses OpenSSH. You can run this code to find the links to the latest releases of OpenSSH:

$url = 'https://github.com/PowerShell/Win32-OpenSSH/releases/latest/'

$request = [System.Net.WebRequest]::Create($url)

$request.AllowAutoRedirect=$false

$response=$request.GetResponse()

$([String]$response.GetResponseHeader("Location")).Replace('tag','download')

+ '/OpenSSH-Win64.zip'

$([String]$response.GetResponseHeader("Location")).Replace('tag','download')

+ '/OpenSSH-Win32.zip'

You’ll see results like this:

https://github.com/PowerShell/Win32-OpenSSH/releases/download/v0.0.17.0/

OpenSSH-Win64.zip

https://github.com/PowerShell/Win32-OpenSSH/releases/download/v0.0.17.0/

OpenSSH-Win32.zip

Download the appropriate version. Instructions for installing a Windows version of OpenSSH are available at <http://mng.bz/n48S>. The instructions don’t explicitly state it, but ensure that all instances of PowerShell or CMD are started with elevated privileges when installing OpenSSH. When OpenSSH is installed, perform the additional configuration steps for Windows machines at <http://mng.bz/10iL>.

OpenSSH is available for most Linux distributions. Install, or update, both client and server versions of OpenSSH and configure as described at the SSHRemoting URL given earlier.

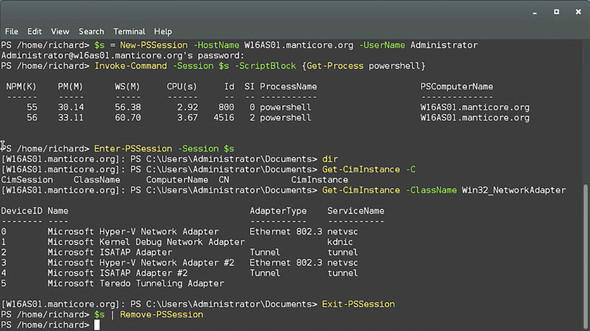
You also need to ensure that the Linux and Windows machines can find each other on the network. Either ensure that your DNS contains entries for all relevant machines or add appropriate entries to the hosts file on your machines.

PowerShell remoting from Linux to Windows works in a similar manner to Windows to Windows remoting. An example of a remoting session from a Linux machine to a Windows machine is shown in [figure 5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-161.xhtml#app01fig05).

**Note**

PowerShell remoting between Linux and Windows machines works only with PowerShell v6 because it’s the only version that supports the use of SSH.

Figure 5: Remoting session from Linux to Windows



Create a remoting session from the Linux machine to the Windows machine:

$s = New-PSSession -HostName W16AS01.manticore.org

-UserName Administrator

You’ll be prompted for the password of the user account you specify. Invoke-Command can then be used, as with the remoting sessions you’ve already seen:

Invoke-Command -Session $s -ScriptBlock {Get-Process}

A PowerShell remoting session from Linux to Windows can also be used interactively. Notice the prompt change in [figure 5](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/section-161.xhtml#app01fig05). Within the remoting session, you can access the functionality on the Windows machine that’s not present on the Linux machine, such as the CIM cmdlets.

You can also create a remoting session from Windows to Linux:

PS> $sl = New-PSSession -HostName Lin01.manticore.org -UserName root

root@lin01.manticore.org's password:

You’ll be prompted for the password. Commands can be run over the session:

PS> Invoke-Command -Session $sl -ScriptBlock {Get-Process -Name powershell}

NPM(K) PM(M) WS(M) CPU(s) Id SI ProcessName PSComputerName

------ ----- ----- ------ -- -- ----------- --------------

0 0.00 0.02 11.21 5863 5809 powershell Lin01.manticore.org

0 0.00 0.02 4.07 7898 7898 powershell Lin01.manticore.org

or you can enter the session:

PS> Enter-PSSession -Session $sl

[Lin01.manticore.org]: PS /root> $PSVersionTable

Name Value

---- ----.

Name Value

---- ----.

PSVersion 6.0.0-beta

PSEdition Core

GitCommitId v6.0.0-beta.3

OS Linux 3.10.0-514.6.1.el7.x86\_64 #1 SMP Wed Jan 18 13:06:36 UTC 2017

Platform Unix

PSCompatibleVersions {1.0, 2.0, 3.0, 4.0...}

PSRemotingProtocolVersion 2.3

SerializationVersion 1.1.0.1

WSManStackVersion 3.0

[Lin01.manticore.org]: PS /root> Exit-PSSession

You can copy a file from a Windows machine to a Linux machine:

PS> Copy-Item -Path .\test.txt `

-Destination "/home/richard/Scripts/" -ToSession $sl -Force

and vice versa:

PS> Copy-Item -Path "/home/richard/Scripts/\*.txt" `

-Destination .\PIA3e\ -FromSession $sl -Force

You can even use a WS-MAN-based session to a Windows machine and a SSH session to a Linux machine together:

PS> $sw = New-PSSession -ComputerName W16DSC01

PS> Get-PSSession

Id Name ComputerName ComputerType State ConfigurationName

-- ---- ------------ ------------ ----- -----------------

1 SSH1 Lin01.ma... RemoteMachine Opened DefaultShell

2 WinRM2 W16DSC01 RemoteMachine Opened Microsoft.PowerShell

PS> Invoke-Command -Session $sl, $sw `

-ScriptBlock {Get-Process -Name PowerShell}

NPM(K) PM(M) WS(M) CPU(s) Id SI ProcessName PSComputerName

------ ----- ----- ------ -- -- ----------- --------------

27 53.44 62.52 1.22 1740 2 powershell W16DSC01

0 0.00 0.02 12.18 5863 5809 powershell Lin01.manticore.org

0 0.00 0.02 7.70 7898 7898 powershell Lin01.manticore.org

PowerShell remoting between Linux and Windows machines enables you to perform your administration on whichever platform you prefer. Whatever your mix of Linux and Windows machines, you can administer them using the same PowerShell tools.

**Note**

PowerShell v6 remoting over SSH is a possible answer to the issue of accessing non-domain Windows machines remotely. The use of SSH bypasses the Kerberos-related issues that make non-domain remoting difficult and is an alternative to the use of certificate-based remoting, which is the current recommendation.

In [chapter 18](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-18-135.xhtml#ch18) we showed you how to use DSC. What we didn’t cover was that DSC is also available for Linux machines.

## DSC and Linux

The agent side of DSC for Linux has been available since PowerShell v4. You need to install a number of pre-requisite packages on the Linux target machine to support the DSC for Linux agent. Also, you’ll need to download the modules from the PowerShell gallery that provide the resources for configuring Linux.

The client side—Start-DSCConfiguration—isn’t going to be supported in PowerShell v6. You can compile configurations on Linux but you can’t use any of the DSC cmdlets because they’re CIM based.

#### Note

DSC agent for Linux isn’t dependent on PowerShell 6.0, but we’re combining the Linux-based material into this appendix for ease of reference.

We briefly cover installing DSC for Linux and then show you a DSC for Linux configuration.

### Installing DSC for Linux

Installing the DSC agent on Linux is a multi-stage process. First, you install the Open Management Infrastructure (OMI), which is a CIM server for Linux. Then you ensure that prerequisite packages are installed, and then you can install the DSC package on the Linux machine. The final step is to download the DSC resource modules to your Windows authoring machine.

DSC uses CIM, as we showed in [chapter 18](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-18-135.xhtml#ch18). Linux doesn’t have a native CIM provider, so you need to install OMI.

#### Install OMI on a Linux Machine

OMI is an open source project—<https://github.com/Microsoft/omi>—to develop a portable and highly modular CIM Object Manager (CIMOM). It can be built and installed on most UNIX and Linux systems. It’s also used in network switches, including those from Arista and Cisco.

Before you install OMI, ensure you have OpenSSL—at least version 0.9.8 and preferably 1.0.x—on the Linux system. Download OMI from <https://github.com/Microsoft/omi/releases> and install. You should also download and install the appropriate package for your system from <http://mng.bz/SPAg> so that OMI can use the PowerShell Remoting Protocol over WS-MAN. With this package installed, you can create CIM sessions to the Linux system from a Windows machine.

You should check that OMI is running by using the following:

sudo /opt/omi/bin/omicli ei root/omi OMI\_Identify

You’ll see a listing of all instances of the OMI\_identify class in the root/omi namespace.

Now it’s time to install DSC for Linux.

#### Install DSC for Linux on a Linux Machine

DSC for Linux is also an open source project at <https://github.com/Microsoft/PowerShell-DSC-for-Linux>. The pre-requisites for installing DSC for Linux (descriptive name and package name) are as follows:

* GNU C Library—glibc
* CURL http client library—libcurl
* Python—python
* Python Ctypes library—python-ctypes
* Open Management Infrastructure—omi
* OpenSSL libraries—openssl

The installation package for DSC for Linux is available from <http://mng.bz/7Z39>. Download and install. The last step is to download the DSC resource modules.

#### Install DSC for Linux Module on a Windows Machine

If you remember from [chapter 18](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-18-135.xhtml#ch18), DSC resources are used to create configurations. The DSC resources for configuring Linux are available on the PowerShell gallery. Their names start with the prefix nx:

PS> Find-Module nx\* | Format-Table Version, Name, Description

Version Name Description

------- ---- -----------

1.0 nx Module with DSC Resources for Linux

1.1 nxNetworking Module with DSC Networking Resources for Linux

1.1 nxComputerManagement Module with DSC Computer Management Resources

for Linux

You can install all three modules in one pass:

PS> Find-Module nx\* | Install-Module -Force

The DSC resources currently available for configuring Linux include the following:

PS> Get-Module -ListAvailable nx\* |

foreach {Get-DscResource -Module $\_.Name} |

Format-Wide -Column 4

nxArchive nxEnvironment nxFile nxFileLine

nxGroup nxPackage nxScript nxService

nxSshAuthorizedKeys nxUser nxComputer nxDNSServerAddress

nxFirewall nxIPAddress

Linux administration is performed by configuring the contents of numerous files. If a resource isn’t available to configure a particular aspect of your Linux machine, you should be able to complete the task by modifying the contents of the appropriate file.

Before attempting to configure the Linux machine, you should test that you can connect to OMI.

#### Test CIM on a Linux System

The easiest way to test connectivity to CIM on your target machine is to create a CIM session to that machine:

PS> $cred = Get-Credential root

PS> $sopt = New-CimSessionOption -UseSsl -SkipCACheck `

-SkipCNCheck -SkipRevocationCheck

PS> $sl = New-CimSession -Credential $cred -Authentication Basic `

-ComputerName Lin01 -SessionOption $sopt

Create a PowerShell credential object for the root account on the Linux system. You then need to create a set of options for the CIM session. In this case, you’re telling the system to use SSL (encrypt the connection) but to skip all the tests on the machine’s SSL certificate. You can then create the session using the credential and options you set earlier and configuring the session to use Basic (user name/password) authentication.

The resultant CIM session looks identical to a similar session established to a Windows machine:

PS> $sl

Id : 1

Name : CimSession1

InstanceId : 6dd1b519-db6e-4fbf-b26e-91b86bcb79e7

ComputerName : Lin01

Protocol : WSMAN

OMI doesn’t install any useful classes for configuring your Linux machine directly, but you can display some basic information as a test:

PS> Get-CimInstance -CimSession $sl -ClassName OMI\_Identify `

-Namespace root/omi

InstanceID : 2FDB5542-5896-45D5-9BE9-DC04430AAABE

SystemName : Lin01

ProductName : OMI

ProductVendor : Microsoft

ProductVersionMajor : 1

ProductVersionMinor : 1

ProductVersionRevision : 0

ProductVersionString : 1.1.0-0

Platform : LINUX\_X86\_64\_GNU

OperatingSystem : LINUX

Architecture : X86\_64

Compiler : GNU

ConfigPrefix : GNU

ConfigLibDir : /opt/omi/lib

ConfigBinDir : /opt/omi/bin

ConfigIncludeDir : /opt/omi/include

ConfigDataDir : /opt/omi/share

ConfigLocalStateDir : /var/opt/omi

ConfigSysConfDir : /etc/opt/omi/conf

ConfigProviderDir : /etc/opt/omi/conf

ConfigLogFile : /var/opt/omi/log/omiserver.log

ConfigPIDFile : /var/opt/omi/run/omiserver.pid

ConfigRegisterDir : /etc/opt/omi/conf/omiregister

ConfigSchemaDir : /opt/omi/share/omischema

ConfigNameSpaces : {root-omi, interop, root-Microsoft-

DesiredStateConfiguration, root-Microsoft-Windows-

DesiredStateConfiguration}

PSComputerName : Lin01

This is pretty much the same information you saw when you tested that OMI was running from the Linux machine.

Now it’s time to create a configuration.

### Using DSC for Linux

Using DSC to configure a Linux machine is the same as configuring a Windows machine:

* Create a configuration file
* Create a MOF file from the configuration
* Apply the MOF file to the target machine

The configuration file is first.

#### Creating a Configuration File

A configuration file for a Linux machine is identical to that for a Windows machine, except that a different set of resources, defined in the nx\*, modules must be used. We’ll repeat our first configuration from [chapter 18](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-18-135.xhtml#ch18)—to create a file and set its contents as an example, as shown in the following listing.

#### Listing 2: DSC for a Linux configuration file

Configuration LxDSCConfig

{

param ([string]$node)

Import-DSCResource -Module nx

Node $node

{

nxFile myTestFile

{

Ensure = "Present"

Type = "File"

DestinationPath = "/tmp/dsctest"

Contents="This is our DSC on Linux Test!"

}

}

}

LxDSCConfig -node Lin01 -OutputPath .\MOF

The configuration starts with the Configuration keyword and the configuration name. A single parameter, $node, is accepted by the configuration. The module containing the Linux DSC resources is imported, and the nxFile resource is used to configure the file and its contents.

A MOF file is created in the location defined by -OutputPath. You can now apply the configuration.

#### Applying a Configuration

Pushing a configuration to a Linux machine is identical to pushing to a Windows machine if you’re pushing the configuration from a Windows machine. Start-DSCConfiguration isn’t supported on Linux yet and won’t be for the PowerShell v6 release.

#### Note

You can compile configurations into MOF files on Linux—you just can’t deploy them from the Linux machine.

Let’s push the configuration to our Linux machine:

PS> Start-DscConfiguration -CimSession $sl -Path .\MOF\ -Verbose -Wait

VERBOSE: Perform operation 'Invoke CimMethod' with following parameters,

''methodName' = SendConfigurationApply,'className' =

MSFT\_DSCLocalConfigurationManager,'namespaceName' = root/Microsoft/Windows/

DesiredStateConfiguration'.

VERBOSE: Operation 'Invoke CimMethod' complete.

VERBOSE: Time taken for configuration job to complete is 0.655 seconds

You’ll need to use a CIM session (the one we created earlier) to push your configuration. You can test the configuration:

PS> Test-DscConfiguration -CimSession $sl

True

and view the configuration:

PS> Get-DscConfiguration -CimSession $sl

DestinationPath : /tmp/dsctest

SourcePath :

Ensure : present

Type : file

Force : False

Contents : This is our DSC on Linux Test!

Checksum :

Recurse : False

Links : follow

Group : root

Mode : 644

Owner : root

ModifiedDate : 23/02/2017 20:09:34

PSComputerName : Lin01

CimClassName : MSFT\_nxFileResource

DSC for Linux performs and operates in the same way as DSC against Windows that you saw in [chapter 18](https://cdn2.percipio.com/1650106313.ccf6b552e8eedc457fc3dc4bd7ac44556e0c7780/eod/books/147127/OEBPS/chapter-18-135.xhtml#ch18). You can also configure a Linux machine to utilize a pull server and even mix and match Linux and Windows configurations in the same file.

#### Note

Hint: use roles to separate the two types of machine.

## Summary

* PowerShell core engine is now an open source project.
* Powershell v6 will be available for Windows, Linux, and macOS.
* PowerShell v6 is built on .NET Core.
* PowerShell remoting uses SSH for Windows to Linux or Linux to Windows connections. Windows to Windows can use WS-MAN or SSH.
* PowerShell for Linux is still in its infancy compared to Windows PowerShell but is already capable of performing basic management tasks on your systems.
* The porting of PowerShell and DSC to Linux means you can manage your heterogenous environments with a single set of tools.
* Windows Powershell will continue to be the version that ships OOB with Windows and will continue to evolve through its ecosystem of modules.