2

**Introducing PowerShell 7**

This chapter covers the following recipes:

* Exploring new Operators
* Exploring Parallel processing
* Explore Performance improvements
* Using Test-Connection
* Using Select-String
* Exploring Error view and Get-Error

# Introduction

In Chapter 1, you installed and configured PowerShell 7, along with VS code and a new font. The recipes enabled you to get PowerShell 7 installed and configured. In this chapter, we look at PowerShell 7 and how it differs from Windows PowerShell. The recipes in this chapter illustrate some of the important new features which come with PowerShell 7.

PowerShell 7 is cross-platform, and you can install it into Linux and macOS. With PowerShell now being cross-platform, there is a new audience to PowerShell, one with a background in Linux shells such as Bash. As a result, the PowerShell team added several new operators.

In Windows PowerShell, the Foreach syntax item and the Foreach-Object command allowed you to process collections of objects. With Windows PowerShell, each iteration through a collection was serial, which could result in a very long script run times PowerShell 7 introduces an improvement in the Foreach‑Object command that enables you to run iterations in parallel.

With the move to open source, the PowerShell code was open to inspection by the community. Many talented developers were able to make improvements to the performance, particularly in how the Foreach syntax element works.

Another significant improvement in PowerShell 7 is the update to Test-Connection, a command you use to test a network connection with a remote system. Test-Connection, in PowerShell 7 not only does more but is significantly faster than with Windows PowerShell.

Error reporting in Windows PowerShell was excellent. Clear and generally actionable error messages with details of exactly where the error occurred.

# Exploring New Operators

Operators are symbols or combinations of keystrokes which PowerShell recognizes and assigns some meaning. PowerShell uses the ‘+’ operator to mean addition, either arithmetic addition or string addition/concatenation. Most of the PowerShell operators were defined with Windows PowerShell V1.

PowerShell 7 now implements some new operators, including:

* Pipeline Chain
* Null-coalescing operator ??
* Null conditional assignment operator ??=
* Experimental Null conditional member access operators ?. and ?[]
* Background Processing Operator &

## Getting Ready

This recipe uses SRV1, a Windows Server 2020 host. You have installed and configured PowerShell 7 and VS Code.

## How to do it...

1. Checking results traditionally

Write-Output 'Something that succeeds'

if ($?) {Write-Output 'It worked'}

1. Check results with pipeline operator &&

Write-Output 'Something that succeeds' && Write-Output 'It worked'

1. Using Pipeline chain operator  ||

Write-Output 'Something that succeeds' ||

Write-Output 'You do not see this message'

1. Define a simple function

function Install-CacadiaPLFont{

  Write-Host 'Installing Cascadia PL font...'

}

1. Demonstrate || operator

$OldErrorAction        = $ErrorActionPreference

$ErrorActionPreference = 'SilentlyContinue'

Get-ChildItem -Path C:\FOO\CASCADIAPL.TTF | OUT-NULL ||

   Install-CacadiaPLFont

$ErrorActionPreference = $OldErrorAction

1. Create a function to test null handling

Function Test-NCO {

  if ($args -eq '42') {

    Return 'Test-NCO returned a result'

  }

}

1. Test null results traditionally

$Result1 = Test-NCO    # no parameter

if ($null -eq $Result1) {

    'Function returned no value'

} else {

    $Result1

}

$Result2 = Test-NCO 42  # using a parameter

if ($null -eq $Result2) {

    'Function returned no value'

} else {

    $Result2

}

1. Test using Null Coalescing operator ??

$Result3 =  Test-NCO

$Result3 ?? 'Function returned no value'

$Result4 =  Test-NCO 42

$Result4 ?? 'This is not output, but result is'

1. Demonstrate Null Conditional Assignment Operator

$Result5 = Test-NCO

$Result5 ?? 'Result is is null'

$Result5 ??= Test-NCO 42

$Result5

1. Test running a method on a null object traditionally

$BitService.Stop()

1. Show Null conditional operator for a method

${MyService}?.Stop()

1. Test Null property name access

$x = $null

${x}?.propname

$x = @{Propname=42}

${x}?.propname

1. Test array member access if a null object

$y = $null

${y}?[0]

$y = 1,2,3

${y}?[0]

1. Use background processing operator &

Get-CimClass -ClassName Win32\_Bios &

1. Get the results of the job

$JobId = (Get-Job | Select -last 1).Id

Wait-Job -id $JobId

$Results = Receive-Job -Id $JobId

$Results |

  Get-Member |

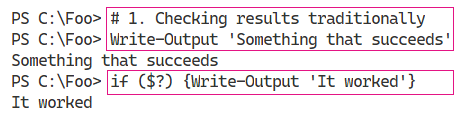
    Select-Object -First 1

1. View the output

$Results | Format-Table

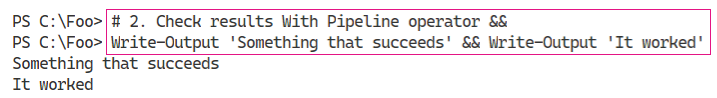
## How it works...

In step 1, you write output, which succeeds. Then you test the value of $? to determine whether that previous step did, in fact, succeed. The output is as follows:



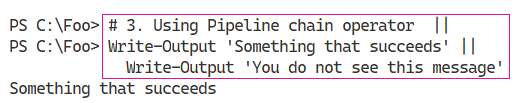
Insert image B42024\_02\_01.png

In step 2, you use the && operator to check that a proceeding command had finished without an error. The output looks like this



Insert image B42024\_02\_02.png

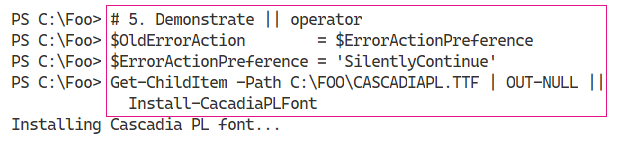
The pipeline chain operator, ||, tells PowerShell to run the commands after the operator if he preceding command succeeded. In step 3, you see the operator in use, with output like this:



Insert image B42024\_02\_03.png

In step 4, you define a function. Defining the function produces no output. This function writes output, to simulate the installation of the Cascadia Code PL font.

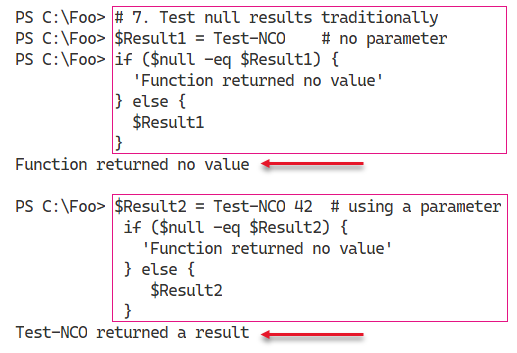
In step 5, you check to see if the TTF file exists, and if not, you call the Install-CascadiaPLFont function to simulate installing the font. The output of this snippet looks like this:



Insert image B42024\_02\_04.png

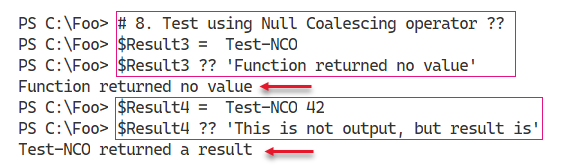
To illustrate the handling of null results from a function, in step 6, you create a function. This function either returns nothing (if you call the function with no parameters) or the function returns a string value if you call it specifying a parameter.

In step 7, you illustrate the traditional handling of a function which returns a null. You call the function, first without a parameter, which returns no result then with a value which does return a value. You then test to see whether the function returned a value in each case, which looks like this:



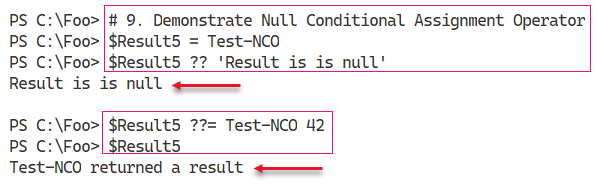
Insert image B42024\_02\_05.png

When you use the null=coalescing operator (??) between two operands, the operator returns the value of its left-hand operand if it isn't null; otherwise, it evaluates the right-hand and returns the results. In step 8, you call the Test-NCO function and check whether or not the function returns a value, which looks like this:



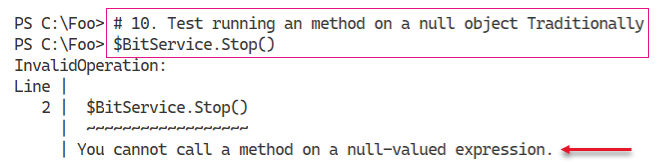
Insert image B42024\_02\_06.png

You use the null conditional assignment operator, ??=, to assign a value to a variable if that variable is currently null as you can see in step 9, the output form which looks like this:



Insert image B42024\_02\_07.png

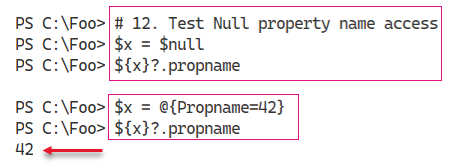
On common issue often seen in the various PowerShell support forums arises when you attempt to invoke a method on an object which is null. You might have used an expression or a command to attempt to return a value (for example all AD users in the Marin County office) and which produces a null. In Step 10, you attempt to invoke the Stop() method on the $BitSService object. Since you have not assigned a value to $BitsService, you see result (an error “You cannot call a method on a null-valued expression”). This traditional error looks like:



Insert image B42024\_02\_08.png

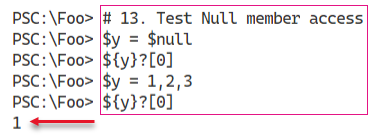
By using the null conditional operator, you can run the Stop() method if the $BitsService variable is non-null, but skips calling the method if the variable is null. In effect, what you are doing in step 11 is to call the Stop() method if the variable is non-null, and do nothing otherwise. Because the variable does not have a value, this step does nothing (and produces no output).

When a variable is null (whether due to an error in your scripts, or because a command returns a null instead of an expected value), accessing property names can also cause errors. As shown in step 12, if an object has no properties (named propname), you can avoid an error message that would have produced in Windows PowerShell. The output of step 12, in PowerShell 7, looks like this:



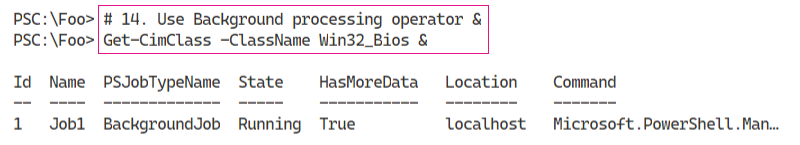
Insert image B42024\_02\_09.png

You can also encounter issues with null objects when you attempt to access an array member of an object that may or may not exist. In step 13, you attempt to access an array member of an array that does not exist, and then does exist, which looks like this:



Insert image B42024\_02\_10.png

In step 14, you investigate the use of the background processing operator &. The idea is that you append this character to the end of a command or script, and PowerShell runs that code in the background. The output from this step looks like this:



Insert image B42024\_02\_11.png

## There's more...

In step 4, the function you create simulates the installation of a font if the font does not exist. The font file, CASCADIAPL.TTF, is a TrueType font file for the Cascadia Code Powerline font. This is the Cascadia Code font you installed in Chapter 1 with the addition of symbols for Powerline. For more details on this font, see: https://www.hanselman.com/blog/PatchingTheNewCascadiaCodeToIncludePowerlineGlyphsAndOtherNerdFontsForTheWindowsTerminal.aspx.

In step 5, you simulate installation of the font if it does not already exist. When you check to test whether the TTF file currently exists, the default setting for $ErrorActionPreference (Continue) means you would see an error message if the file does not exist. To avoid the error message, the snippet sets the value of $ErrorActionPreference to SilentlyContinue and after ensuring the

In step 11 and step 12, you attempt to access a property from an object. The assumption here is that you only want to invoke the method or access a property value if the object exists and you do not care otherwise. Thus, if $BitsAdmin has a value, you call the Stop() method, otherwise, the code carries on without the script generating errors. This approach is great from the command line, but in production scripts, the approach could mask other underlying issues. As with all PowerShell features, you have to use null handling with due care and attention.

With step 14, you tell PowerShell to run a command as a background job by appending the & character to the command. This is a simpler way to invoke the command as a job than by calling Invoke-Command specifying the command using the -ScriptBlock or -Script parameters.

In step 15, you use Get-Job, Wait-Job and Receive-Job to wait for the last job run and get the output. One downside to not using Start-Job to create a background is that you can not specify a job name. That means using the technique shown in step 15 to obtain the job and job results to the job create in step 14. Using that technique is thus more useful from the command line than in a production script.

In this recipe, you have seen the new operators added to PowerShell 7. Most of them provide a short cut way to do something, particularly at the command line.

# Exploring Parallel Processing

There are often times with PowerShell when you want to run a number of commands in parralell. For example, you might have a list of computer names. For each of those computers, you want to run a script on that computer, for example to test that various services are running on that computer. In this scenario, you might use Get-Content to get an array of computer names, then use either Foreach or Foreach-Object to run the script on the computer. If there are 10 computers and the script takes 10 minutes, the total run time is over 100 minutes.

With Windows PowerShell, the only built in methods of running scripts in parallel could be done by using background jobs or by using workflows. With background jobs, you could create a set of jobs each of which starts some script on a single computer. In that case, PowerShell runs each job in a separate process which provides isolation between each job but is resource intensive. Workflows, added with Windows PowerShell V4 also allowed you to run script blocks in parallel. However, workflows are not carried forward into PowerShell 7, and you may find difficulty in converting some script blocks to workflows.

An alternative to background jobs is to use the ThreadJob module you can download from the PowerShell gallery. For more details on this module, see the module’s repository page at https://github.com/PaulHigin/PSThreadJob.

With PowerShell 7, the PowerShell team added an option to the Foreach-Object which allows you to run script blocks in parallel. This simplifies running script blocks or scripts, especially long running ones, in parallel and avoids needing third party modules or dealing with the complexity of workflows.

This recipe demonstrates running operations in parallel traditionally, using background jobs, and using Foreach-Object -Parallel.

## Getting Ready

You run this recipe on SRV1 after you have installed PowerShell 7 and optionally VS Code.

## How to do it...

1. Simulate a long running script block

$SB1 = {

  1..3 | ForEach-Object {

    "In iteration $\_"

    Start-Sleep -Seconds 5

  }

}

Invoke-Command -ScriptBlock $SB1

1. Time the expression

Measure-Command -Expression $SB1

1. Refactor into using jobs

$SB2 = {

1..3 | ForEach-Object {

  Start-Job -ScriptBlock {param($X) "Iteration $X " ;

                          Start-Sleep -Seconds 5} -ArgumentList $\_

}

Get-Job | Wait-Job | Receive-Job -Keep

}

1. Invoke the script block

Invoke-Command -ScriptBlock $SB2

1. Remove the old jobs and time the script block

Get-Job | Remove-Job

Measure-Command -Expression $SB2

1. Define a script block using ForEach-Object -Parallel

$SB3 = {

1..3 | ForEach-Object -Parallel {

               "In iteration $\_"

               Start-Sleep -Seconds 5

         }

}

1. Execute the script block

Invoke-Command -ScriptBlock $SB3

1. Now measure it

Measure-Command -Expression $SB3

1. Create then run two short  script blocks

$SB4 = {

    1..3 | ForEach-Object {

                   "In iteration $\_"

             }

}

Invoke-Command -ScriptBlock $SB4

$SB5 = {

        1..3 | ForEach-Object -Parallel {

                       "In iteration $\_"

             }

}

Invoke-Command -ScriptBlock $SB5

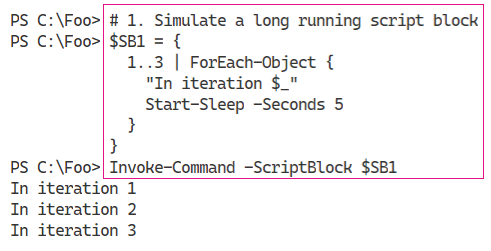
1. Measure time for both

Measure-Command -Expression $SB4

Measure-Command -Expression $SB5

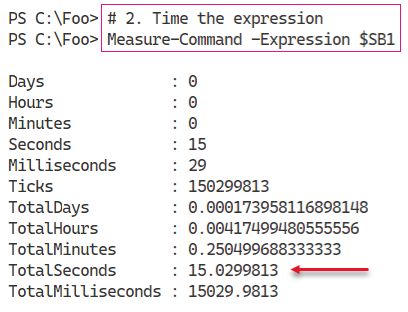
## How it works...

In step 1, you create and then invoke a script block. The script block simulates how you can run several long script blocks using the Foreach-Object in a traditional manner, with output like this:



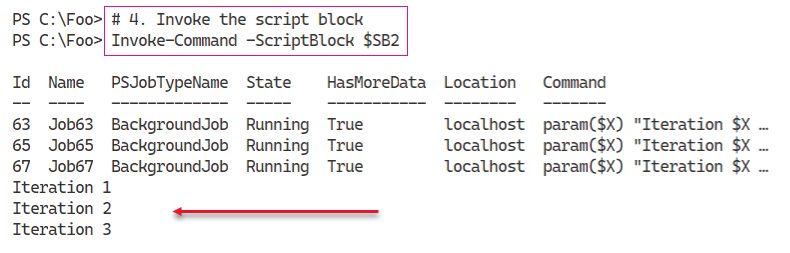
Insert image B42024\_02\_12.png

In step 2, you determine how long it takes PowerShell to run this script block, with output like this:



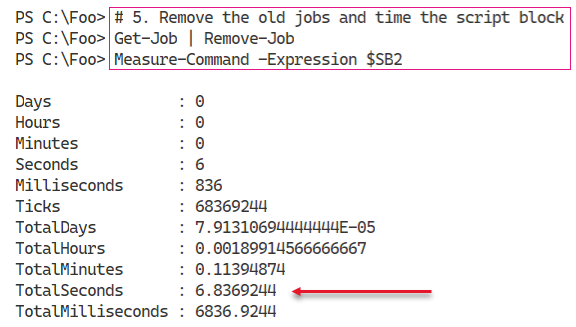
Insert image B42024\_02\_13.png

In step 3, you refactor the $SB1 script block to use PowerShell background jobs. The script block runs the simulated long-running task using jobs then waits for and displays the output from each job. Defining the function creates no output. IN step 4, you invoke the script block to view the results, which looks like this:



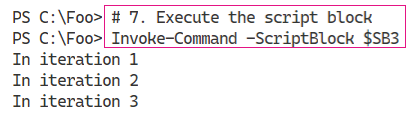
Insert image B42024\_02\_14.png

In step 5, you re-run the updated script block to determine the runtime. The output of this step looks like this:



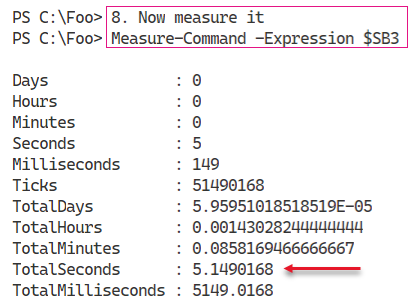
Insert image B42024\_02\_15.png

In step 6, you create a further script block that uses the PowerShell 7 Foreach-Object -Parallel construct. When you define this script block, PowerShell creates no output. In step 7, you run the script block, which looks like this,



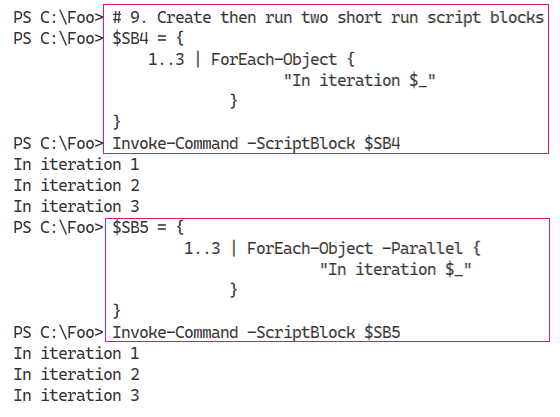
Insert image B42024\_02\_16.png

In step 8, you time the execution of the script block, making use of the Foreach-Object -Parallel feature, which looks like this:



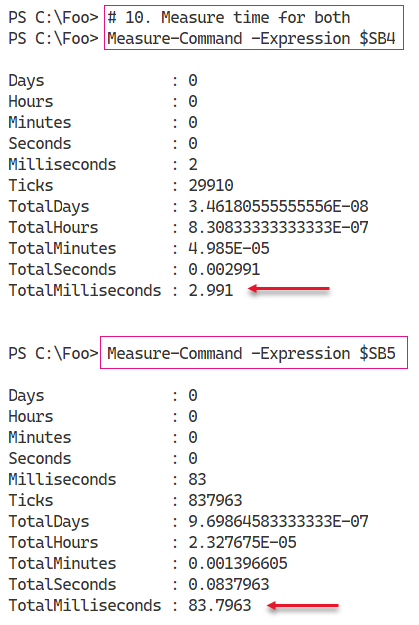
Insert image B42024\_02\_17.png

In step 9, you define and invoke two script blocks, which looks like this:



Insert image B42024\_02\_18.png

In the final step in this recipe, step 10, you measure the execution time of these two script blocks, which looks like this:



Insert image B42024\_02\_19.png

## There's more...

1. In step 1 and step 2, you

There is some overhead underpinning Foreach=Object -Parallel. The command has to, under the covers, create and manage separate threads for each parallel iteration. In some cases, the overhead means that using -Parallel is slower as illust

# Exploring Performance Improvements

In PowerShell 7, certain objects added by the PowerShell 7 installer (and PowerShell 7) differ from those used by Windows PowerShell.

## Getting Ready

This recipe uses SRV1 after you have installed PowerShell 7.

## How to do it...

## How it works...

In step 1,

## Getting Ready

You run this recipe on SRV1 after you have installed PowerShell 7.

## How to do it...

## There's more...

In step 4, you download a sample profile file. This file contains some customizations for the PowerShell console configuration, including changing the default starting folder (to C:\Foo), creating some aliases, PowerShell drives and a credential object. These represent sample content you might consider for including in your console profile file. Note that VIS Code, which you install in the next recipe, uses a separate current user/current host profile file which means you can customize PowerShell at the console and in VS code differently.

# Using Select-String

## Getting Ready

You run this recipe on SRV1 after you have installed PowerShell 7 and have created a console profile file.

## How to do it...

## There's more...

# Exploring Error View and Get-Error

As part of the launch of Visual Studio Code, Microsoft also created a new and free type font which you can download and use both at the PowerShell 7 console and inside VS Code. This recipe shows how you can download the font, install it, and set to be the default in VS Code.

## Getting Ready

## How to do it...

## How it works...

## There's more...