

# PowerShell ForEach-Object and ForEach Loop Explained

PowerShell ForEach and ForEach-Objects loops are one of the most commonly used functions. They allow you to easily iterate through collections. But what is the difference between **ForEach-Object** and the **ForEach()** cmdlets?

Both cmdlets look similar, and basically, they allow you to perform the same action. But knowing the difference between the two, allows you to improve your PowerShell script.

In this article, we are going to take a look at how to use PowerShell ForEach and ForEach-Object and explain the difference between the two.

## ForEach() vs ForEach-Object

The PowerShell **ForEach()** and **ForEach-Object** cmdlet both allow you to iterate through a collection and perform an action against each item in it. However, there is a difference in how they handle the data and in the usability.

The **ForEach()** cmdlet loads the entire collection into the memory before it starts processing the items. By loading the collection into the memory of your computer, it's faster than the **ForEach-Object** cmdlet.

```
foreach ($fruit in $fruitsAvailable) {  
    Write-Host $fruit  
}
```

On the other hand, the PowerShell ForEach-Object cmdlet also you to pipe it behind another cmdlet. This way it can process each item as it comes down the pipeline. This also means that you can pipe other cmdlets behind it.

```
$fruitsAvailable | ForEach-Object {  
    Write-Host $_  
}
```

Besides these two cmdlets, we also have the two aliases **ForEach** (note, without the parenthesis **()**) and **%**. Both are an alias for the **ForEach-Object** cmdlet, but this can be a bit confusing sometimes. The commands below are all the same:

```
$fruitsAvailable | ForEach-Object {  
    Write-Host $_  
}  
$fruitsAvailable | ForEach {  
    Write-Host $_  
}
```

```
$fruitsAvailable | % {  
Write-Host $_  
}
```

Now for a small collection, the performance difference and memory usage are minimal. In the example above is the `ForEach()` statement only 4ms faster and the scriptblock is easier to read. But when working with large collections, you might want to take memory usage into consideration.

When you are loading a large CSV file, for example, it's more efficient to pipe the `ForEach-Object` cmdlet behind it, instead of storing the results first into a variable:

```
Import-Csv -Delimiter ";" -Path c:\temp\EmailAddresses.csv | Foreach-Object {Get-ADUser -Identity $_.Name}
```

## FREE EMAIL SERIES!

### Level Up with PowerShell

---

5 Emails, Endless Skills

## PowerShell ForEach

---

The `ForEach` statement in PowerShell allows you to iterate through an array or a list of items. The syntax of the `ForEach` statement is pretty straightforward:

```
foreach ($item in $collection) {  
# Script to execute with $item  
}
```

As you can see, we can specify the collection that we want to loop through, this can be an array or list, and we specify a variable for iterator. This will contain the value of each item that is inside the collection.

For example, to print out each fruit to the console we can do the following:

```
$fruitsAvailable = @("Apple", "Banana", "Orange", "Grapes", "Mango", "Strawberry",  
"Pineapple")  
foreach ($fruit in $fruitsAvailable) {  
Write-Host $fruit  
}
```

```
PowerShell
PS C:\Users\ruud> $fruitsAvailable = @("Apple", "Banana", "Orange", "Grapes", "Mango", "Strawberry", "Pineapple")
PS C:\Users\ruud> foreach ($fruit in $fruitsAvailable) { Write-Host $fruit }
Apple
Banana
Orange
Grapes
Mango
Strawberry
Pineapple
PS C:\Users\ruud> |
```

## Using Collections

---

We can also use the PowerShell **ForEach** statement with a cmdlet that returns a collection. Most will pipe the **ForEach-Object** cmdlet behind a cmdlet, but that is not always necessary. The advantage of using the **ForEach** statement is that it is easier to read compared to the **ForEach-Object** cmdlet.

For example, the **Get-ChildItem** cmdlet returns all the files and folders from a given path. To iterate through the files we can do the following:

```
foreach ($item in Get-ChildItem -path "c:\temp") {
Write-Host $item.Name
}
```

As you can see, the **Get-ChildItem** returns the file or folder object. Inside the script block, we can access all the properties of the item, in this case, the item name.

## Break and Continue

---

When working with **ForEach** loops in PowerShell you sometimes want to stop the iterations when a certain value is found or skip an item based on its value. To do this we can use the **Break** and **Continue** keywords.

For example, if we want to check if a value is evenly divisible by 7. If not, then we continue to the next item:

```
foreach ($i in 1..100) {
if ($i % 7 -ne 0 ) { continue }
Write-Host "$i is a multiple of 7"
}
```

The **Break** keyword allows you to stop the PowerShell **foreach** loop when a certain condition is met. Now I don't use these often, most of the time you can write better code by using **where-object** cmdlets or using appropriate filters. But below is an example to give you an idea of how you could use the **Break** keyword:

```
foreach ($number in 1..10) {
if ($number -gt 5) {
Write-Host "Exiting the loop because $number is greater than 5"
```

```
break
}
Write-Host "Processing number: $number"
}
```

## Nested ForEach Loops

---

Inside a scriptblock, you can use other foreach statements as well. Try to keep it at a minimum, because when you are using too many nested foreach loops, your code will become harder to read and debug.

Take the following example, we have an array of teams and inside each team, we have an array of players. The outer foreach loop iterates over each team in the `$teams` array. The inner foreach loop then iterates over the players within each team.

```
# Define an array of teams, where each team has an array of players
```

```
$teams = @(
@{
TeamName = "Team A"
Players = @("Player1A", "Player2A", "Player3A")
},
@{
TeamName = "Team B"
Players = @("Player1B", "Player2B", "Player3B")
},
@{
TeamName = "Team C"
Players = @("Player1C", "Player2C", "Player3C")
}
)
```

```
# Nested foreach loop to iterate over teams and players
```

```
foreach ($team in $teams) {
Write-Host "Team: $($team.TeamName)"
foreach ($player in $team.Players) {
Write-Host " Player: $player"
}
}
```

## PowerShell ForEach-Object

---

The PowerShell ForEach-Object cmdlet is used when you want to iterate through items that are streamed from a pipeline. The advantage of `ForEach-Object` is that it starts processing the items as they come down the pipeline and you can also pipe other cmdlets behind it.

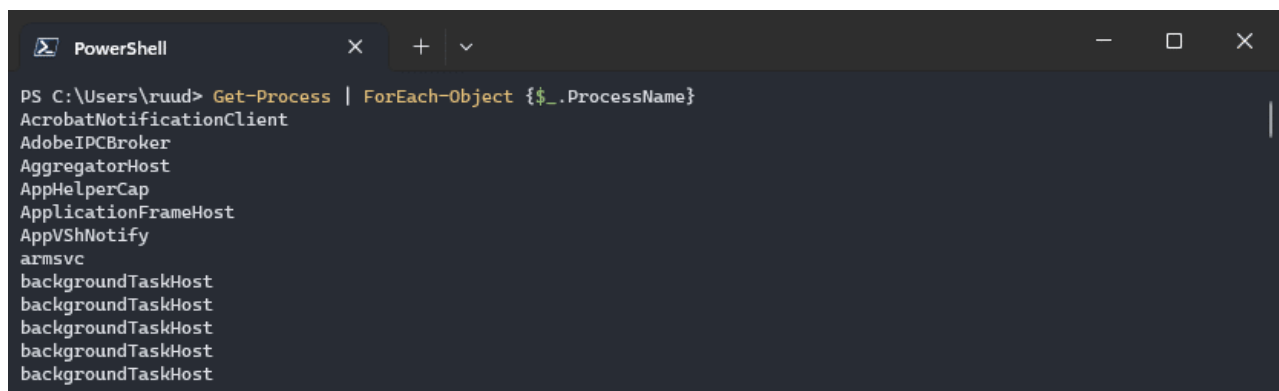
You want to use this cmdlet when you are working with large collections or datasets because even though it's slower than the `foreach` statement, it doesn't consume that much memory.

Another advantage `ForEach-Object` is that you can speed it up by using the parallel switch in PowerShell 7. This allows you to process multiple items simultaneously, but more about that later.

As mentioned, the `ForEach-Object` cmdlet is piped behind another cmdlet. You can use one (or more) script blocks to specify the operation that you want to execute on the current item. For example, to get all process names of the running processes you can do:

```
Get-Process | ForEach-Object {$_.ProcessName}
```

As you can see in the example above, we reference the current item using the `$_` variable.



```
PS C:\Users\ruud> Get-Process | ForEach-Object {$_.ProcessName}
AcrobatNotificationClient
AdobeIPCBroker
AggregatorHost
AppHelperCap
ApplicationFrameHost
AppVShNotify
armsvc
backgroundTaskHost
backgroundTaskHost
backgroundTaskHost
backgroundTaskHost
backgroundTaskHost
```

You can pipe other cmdlets behind the PowerShell `ForEach-Object` scriptblock. So if you want to store each process name in a text file, you could do the following:

```
Get-Process | ForEach-Object {$_.ProcessName} | Out-File "c:\temp\runningprocs.txt"
```

In the examples above, we are executing only a small script in the scriptblock, which allows us to write everything in one single line. But you can also add larger scripts inside the script block:

```
$files = Get-ChildItem -Path "C:\temp\" -File
$files | ForEach-Object {
$fileName = $_.Name
$fileSizeKB = [math]::Round($_.Length / 1KB, 2)
Write-Host "Processing file: $fileName"
Write-Host "File size: $fileSizeKB KB"
Write-Host "Processing complete for $fileName"
}
```

## Begin, End, and Multiple Script blocks

---

When using the `ForEach-Object` cmdlet, you can use a `begin` and an `end` script block. These script blocks are only executed once, whereas the normal scriptblock (the process scriptblock) is executed for each item in the collection.

Begin and End script blocks are great when you need to register the start and end time of a script, or when you want to register the starting or end of a process:

```
1..5 | ForEach-Object -Begin {  
Write-Host "Starting the processing..."  
} -Process {  
$squared = $_ * $_  
Write-Host "Number: $_, Squared: $squared"  
} -End {  
Write-Host "Processing complete."  
}
```

In the example above, we have labeled each script block with what it is, the begin, process, and end block. But you don't need to specify this (I recommend doing it, because it makes your code more readable).

If you specify multiple script blocks, then the first block is always treated as a beginning block.

```
1..2 | ForEach-Object { 'Begin Block' } { 'Process Block' }
```

```
# Result
```

```
Begin Block
```

```
Process Block
```

```
Process Block
```

If you have more than two blocks, then the last is always treated as an end block. Every block in between is a process block.

```
1..2 | ForEach-Object { 'Begin' } { 'Process A Block' } { 'Process B Block' } { 'End' }
```

```
# Result
```

```
Begin
```

```
Process A Block
```

```
Process B Block
```

```
Process A Block
```

```
Process B Block
```

```
End
```

If you want to run multiple process blocks but don't want to use a begin or end block, then you will have to specify the parameters begin, process, and end, and map a `$null` value to the first and latter:

```
1..2 | ForEach-Object -Begin $null -Process { 'Process A Block' } { 'Process B Block' } -
```

```
End $null
```

```
# Result
```

```
Process A Block
```

Process B Block  
Process A Block  
Process B Block

## Using Parallel Processing

---

If you have a really slow script block or when you need to process a lot of items ( 10.000+ ) then you can use the `-parallel` parameter in PowerShell 7.x and higher. This way the script block will run in parallel for each item, allowing you to process multiple items simultaneously.

By default, the parallel parameter will process the items in batches of 5. But with the `-ThrottleLimit` parameter we can define the number of run spaces that we want to use. The example below will process the 8 items in batches of 4:

```
1..8 | ForEach-Object -Parallel {  
Write-Host "Slow script to process $_"  
Start-Sleep 1  
} -ThrottleLimit 4
```

With the parallel parameter, a new runspace is created for each batch. This means that if you need to reference a variable outside the script block, you will need to use the `$using:` variable to reference it:

```
$Message = "Slow script to process:"  
1..8 | ForEach-Object -Parallel {  
Write-Host $using:Message $_  
Start-Sleep 1  
} -ThrottleLimit 4
```

Now good to know is that running scripts in parallel isn't always faster. To be honest, 98% of the time isn't faster. The problem with executing your scriptblock in parallel, is that it takes time to create the runspace to execute the script.

Without the parallel function, your scriptblock is executed in your current PowerShell thread. This way it has access to all your variables, pipeline, and loaded memory. But to execute the script block in parallel, it will need to start a new PowerShell thread (create a new runspace), and this takes time (up to 1 second roughly).

So this only makes sense when your script block is really slow, CPU intensive, or need to wait on an API to respond for example.

For example, writing "Hello World" takes only a couple of milliseconds:

```
Measure-Command { Write-Host "Hello World" } | Select TotalMilliseconds  
# Result  
TotalMilliseconds  
-----  
4,65
```

But when executed in parallel, it will almost take more than 30 milliseconds to execute:

```
Measure-Command { 1 | ForEach-Object -Parallel { Write-Host "Hello World" } } | Select  
TotalMilliseconds
```

# Result

```
TotalMilliseconds
```

-----

31,36

If we however have a script that takes more than 1 second to execute (simulated here with a start-sleep), then the parallel method will have an advantage:

```
Measure-Command { 1..5 | ForEach-Object { Start-Sleep -Seconds 1 } } | Select  
TotalMilliseconds
```

# Result

```
TotalMilliseconds
```

-----

5054,55

# Execute in parallel, with the 5 default threads:

```
Measure-Command { 1..5 | ForEach-Object -Parallel { Start-Sleep -Seconds 1 } } | Select  
TotalMilliseconds
```

# Result

```
TotalMilliseconds
```

-----

1082,24



```
PowerShell
PS C:\Users\ruud> Measure-Command { 1..5 | ForEach-Object { Start-Sleep -Seconds 1 } } | Select TotalMilliseconds
TotalMilliseconds
5054,55
PS C:\Users\ruud> Measure-Command { 1..5 | ForEach-Object -Parallel { Start-Sleep -Seconds 1 } } | Select TotalMilliseconds
TotalMilliseconds
1082,24
PS C:\Users\ruud> |
```

## Break, Continue, and Return

Just like with the foreach statement, we can use **Break** to stop a PowerShell ForEach-Object loop, but we can't use **Continue** though. A break will stop the iteration in the ForEach-Object loop completely and exit the loop:

```
1..10 | ForEach-Object {
```

```
if ($_ -gt 5) {
```

```
Write-Host "Exiting the loop because $_ is greater than 5"
```

```
break
```



```
}  
Write-Host "Processing number: $_"  
}
```

However, if we try to use `Continue` inside a `ForEach-Object` loop, then you will see that it also stops and exists in the loop. So the example below won't work:

```
# This won't work  
1..100 | ForEach-Object {  
if ($_ % 7 -ne 0 ) { continue }  
Write-Host "$_ is a multiple of 7"  
}
```

Instead, we will need to use the `return` keyword to continue to the next item in the collection.

```
1..100 | ForEach-Object {  
if ($_ % 7 -ne 0 ) { return }  
Write-Host "$_ is a multiple of 7"  
}
```



```
PowerShell  
PS C:\Users\ruud> 1..100 | ForEach-Object {  
>> if ($_ % 7 -ne 0 ) { continue }  
>> Write-Host "$_ is a multiple of 7"  
>> }  
PS C:\Users\ruud> 1..100 | ForEach-Object {  
>> if ($_ % 7 -ne 0 ) { return }  
>> Write-Host "$_ is a multiple of 7"  
>> }  
7 is a multiple of 7  
14 is a multiple of 7  
21 is a multiple of 7  
28 is a multiple of 7  
35 is a multiple of 7  
42 is a multiple of 7  
49 is a multiple of 7  
56 is a multiple of 7  
63 is a multiple of 7  
70 is a multiple of 7  
77 is a multiple of 7
```

## Wrapping Up

In most cases, using the `foreach ( )` statement is the fastest and most readable method to use a `ForEach` loop. If you however need to process a lot of data, or need to pipe other cmdlets behind it, then it's better to use the `ForEach-Object` cmdlet.

Keep in mind that the `parallel` function might seem nice to use, but in most cases, it's slower because of the time needed to start up a new runspace. So make sure you measure the differences.

I hope you liked this article, if you have any questions, just drop a comment below. Subscribe to the newsletter if you want to read more of these articles!

Did you **Liked** this **Article**?

Get the latest articles like this **in your mailbox**  
or share this article

I hate spam to, so you can unsubscribe at any time.