Loops and Iterations in PowerShell

WHAT IS A LOOP?

A loop is a programming/scripting language construct, that lets you define arbitrary or a predefined sequence of instructions inside the body of the loop,
that can iterate (repeat) number of times as long as the defined condition is true.

In PowerShell Loops are classified into 7 types -

1. For Loop
2. While loop
3. Do-While loop
4. Do-Until loop
5. Foreach-Object
6. Foreach Statement
7. Foreach Method

FOR LOOP

For loops, also known as **For Statement** are typically used to iterate through a set of commands a specified number of times, either to step through an array or object, or just to repeat the same block of code as needed.

SYNTAX

```
for(<Initialize>; <Condition>; <Repeat>)
{
Command 1
…
…
Command n
}
```

Initialize, condition and repeat are separated by a semi-colon ";" and are wrapped inside a parenthesis '()'

Initialize - You initialize a loop counter(variable) with starting value and it will always be run before the loop begins. For example, if you are running the loop to iterate 10 times, you can initialize it with 0 and end up loop counter at value 9 making it total 10 repetitions.

Condition - At beginning of each iteration of the 'For loop' a condition is tested that results in either a $true or a $false Boolean value. If the condition is $true the body of the loop is executed, and if it is $false then flow of control exits the loop and goes to the next line after the loop.

Repeat - A set of commands run each time the loop repeat, mostly it is used to increment or decrement the loop counter.

BASIC EXAMPLE :

```
For ($i=1; $i -le 10; $i++)  # $i=1 initialize the loop counter with 1 and $i++
increment the value by 1
{
    "10 * $i = $(10 * $i)"   #print the multiplication table of 10, i.e. simply by
multiply $i with 10
}

10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
```

A Repeat placeholder can hold more than one command and each repeat statement is separated by commas. In this case, we have to initialize each loop counter that we will use in repeat placeholder for increment/decrement operation For example:-

```
$j=10
For($i=1;$i -le 10;$i++,$j--)
{
    "Loop Iteration : i=$i and j=$j"
}

RESULT:
Loop Iteration : i=1 and j=10
Loop Iteration : i=2 and j=9
Loop Iteration : i=3 and j=8
Loop Iteration : i=4 and j=7
Loop Iteration : i=5 and j=6
Loop Iteration : i=6 and j=5
Loop Iteration : i=7 and j=4
Loop Iteration : i=8 and j=3
Loop Iteration : i=9 and j=2
Loop Iteration : i=10 and j=1
```

Instead of using increment/decrement command, we can also use multiplication operator and other kind of operations:

```
For($i=2;$i -le 20;$i=$i*2)
{
    $i
}

RESULT:
2
4
8
16
```

Nested Loop

A loop within a loop is known as Nested Loop.

```
For($i=1;$i -le 5;$i++)
{
    For($j=1;$j -le $i; $j++)
    {
        Write-Host "*" -NoNewline
    }
    "`n"
}

RESULT:
*
**
***
****
*****
```

Infinite Loop

All placeholder can also be empty and if there is no condition to test and control the number of iterations of the loop that will result in an **infinite loop** that will keep executing until you break it out by pressing CTRL+C .

```
For(  ;  ;  )
{
"Infinite loop"
}

RESULT:
Infinite loop
Infinite loop
Infinite loop
Infinite loop
```

```
Infinite loop
Infinite loop
Infinite loop
Infinite loop
Infinite loop
Infinite loop
Infinite loop
Infinite loop
Infinite loop
Infinite loop
Infinite loop
Infinite loop
```

WHILE LOOP

Repeats a command or group of commands while a given condition is true.The *while loop tests the loop condition before the first iteration and the set of commands in the loop body are not executed, even once, if the loop condition doesn't match, therefore it is also called as a **Pretest Loop** . Since, flow of loop is getting controlled at the beginning of loop because of condition, so we can say it **Entry Controlled Loop**. Unlike for loop, the condition can only contain a boolean expression enclosed in parenthesis '()'. In while loop, you have to initialize the loop counter before the while statement and increment or decrement statement is defined at the end of the loop body.

SYNTAX

```
Initializing
While(condition)
{
    Command1
    ...
    Commandn
    Increment/decrement
}
```

BASIC EXAMPLE

```
$i=1  #initialize loop counter
While ($i -le 10)    #while loop condition
{
    $i
    $i++   #increment loop counter
}

RESULT:
1
2
```

```
3
4
5
6
7
8
9
10
```

You can also call cmdlets and assign values in the loop condition. Below example can explain this concept

```
while(($inp = Read-Host -Prompt "Select a command") -ne "Q"){
switch($inp){
   L {"File will be deleted"}
   A {"File will be displayed"}
   R {"File will be write protected"}
   default {"Invalid entry"}
   }
}
```

In some situations you may need to exit a loop early based on something other than the loops condition. In this case the **Break** keyword can be invoked in order to exit out of the loop. This final example shows the same functionality, but uses an infinite loop and the Break keyword to exit out at the appropriate time.

```
$i=1
While ($true)
    {
    $i
    $i++
    if ($i -gt 10) {
        Break
        }
    }

RESULT:
1
2
3
4
5
6
7
8
9
10
```

DO-WHILE LOOP

Do-While loop is just like a while loop, executes the block of statements while the condition evaluates to Boolean $true , but a small difference that is, the body of Do-While is executed first then the condition is tested at the end of the loop compared to a While loop. Therefore , we termed this loop as **Posttest Loop** also.

Since the condition is tested at the end a Do-While() loop is an "Exit controlled loop", that means even if the Condition evaluates to Boolean $false a Do-While loop iterates at least once, because condition is tested after the body of the loop is executed.

SYNTAX

```
Do
{
Command sequence
}
While (<condition>)
```

Let see an example to understand the beaviour of do-while loop

SCRIPT:

```
$i = 1
Do
{
    "Loop Iteration i=$i"
    $i++
}
While($i -le 10)

RESULT:
Loop Iteration i=1
Loop Iteration i=2
Loop Iteration i=3
Loop Iteration i=4
Loop Iteration i=5
Loop Iteration i=6
Loop Iteration i=7
Loop Iteration i=8
Loop Iteration i=9
Loop Iteration i=10
```

Let use the calculator example, with the Do-While which actually simplifies the logic and reduces few lines of code

```
$a = 3; $b = 2
Do
{
    "`$a = $a `$b = $b"
    $Choice = Read-Host "1. Add`n2. Substract`n3. Multiply`n4. Divide`n5.
Exit`nChoose a number [1-5]"
    switch($Choice)
    {
        1 {$a+$b}
        2 {$a-$b}
        3 {$a*$b}
        4 {$a/$b}
        5 {Write-Host "Exiting the Loop" -ForegroundColor Yellow;exit}
        Default {Write-Host "Wrong choice, Try again" -ForegroundColor Red}
    }
}
While($Choice -ne 5)

RESULT:

PS > $a = 3 $b = 2
1. Add
2. Substract
3. Multiply
4. Divide
5. Exit
Choose a number [1-5]: 1
5
$a = 3 $b = 2
1. Add
2. Substract
3. Multiply
4. Divide
5. Exit
Choose a number [1-5]: 3
6
$a = 3 $b = 2
1. Add
2. Substract
3. Multiply
4. Divide
5. Exit
Choose a number [1-5]: 4
1.5
$a = 3 $b = 2
1. Add
2. Substract
3. Multiply
4. Divide
5. Exit
Choose a number [1-5]: 7
Wrong choice, Try again
$a = 3 $b = 2
```

```
1. Add
2. Substract
3. Multiply
4. Divide
5. Exit
Choose a number [1-5]: 5
Exiting the Loop
```

Even if we enter 5 first as an input, it will execute atleast once as per its behaviour i.e. execute the 5th case and then exit from the loop

```
$a = 3; $b = 2
Do
{
    "`$a = $a `$b = $b"
    $Choice = Read-Host "1. Add`n2. Substract`n3. Multiply`n4. Divide`n5.
Exit`nChoose a number [1-5]"
    switch($Choice)
    {
        1 {$a+$b}
        2 {$a-$b}
        3 {$a*$b}
        4 {$a/$b}
        5 {Write-Host "Exiting the Loop" -ForegroundColor Yellow;exit}
        Default {Write-Host "Wrong choice, Try again" -ForegroundColor Red}
    }
}
While($Choice -ne 5)

RESULT:

PS > $a = 3 $b = 2
1. Add
2. Substract
3. Multiply
4. Divide
5. Exit
Choose a number [1-5]: 5
Exiting the Loop
```

Difference Between While and Do-While Loop

While Loop	**Do-While Loop**
Entry controlled loop	Exit controlled loop

While Loop	Do-While Loop
Tests the condition before execution of first iteration(Pretest Loop)	Test the condition after execution of first iteration(Posttest Loop)
Loop is not executed when condition evaluates to false	Loop is executed for atleast once even condition evaluates to false
Do not use any other keyword except while	Use do keyword at starting of loop body and while keyword with condition at the end of loop

DO-UNTIL LOOP

Do-Until loops have similar syntax to Do-While, both begin with the Do keyword prefacing a script block, followed by the condition keyword (While or Until) and the condition enclosed in parenthesis '('), but stop processing further, once the condition statement is met. As an example the following two loops function identically, only the condition is reversed:

SYNTAX

```
Do
{
Command sequence
}
Until (<condition is true>)
```

Let see an example to understand the behaviour of Do-Until loop

SCRIPT:

```
$i = 1
Do
{
    "Loop Iteration i=$i"
    $i++
}
Until($i -gt 10)

RESULT:
Loop Iteration i=1
Loop Iteration i=2
Loop Iteration i=3
Loop Iteration i=4
Loop Iteration i=5
Loop Iteration i=6
Loop Iteration i=7
Loop Iteration i=8
Loop Iteration i=9
Loop Iteration i=10
```

Difference b/w Do-While and Do-Until

Do-While Loop	Do-Until Loop
Use while keyword for condition	Use Until keyword for condition
Continue execution while condition is true	Continue execution until condition is true

FOREACH LOOP

Foreach statement or "Foreach loop" is PowerShell scripting language construct that is used to iterate through values in a collection of items,like an Array and perform defined operation against each item.

How PowerShell ForEach Iteration Works

To iterate through a collection of items, you take the first item in the collection and performs an action. When you complete the task on the first item, you take the next item and perform the same action on the item. Then You continue to go back until you have gone through all the items in the collection.

SYNTAX

The syntax of a PowerShell ForEach construct is shown below:

```
ForEach (Item In Collection)
{
ScriptBlock or sequence of commands
}
```

- **Item** : The variable to hold the current item.
- **Collection** : A collection of objects, like ComputerName or Files. Collections will be evaluated and stored in memory before the ScriptBlock is executed.
- **ScriptBlock** : A sequence of commands to be executed against each item in the collection.

The construct of a 'ForEach PowerShell' loop is very straight forward. It simply says: ForEach $item in the $collection, perform the task(s) enclosed in the '{}' block.

The entire Foreach statement must appear on a single line to run it as acommand at the Windows PowerShell command prompt. The entire Foreach statement does not have to appear on a single line if you place the command in a .ps1 script file instead.

Simplest use case is traversing the items or elements of an array, like in the following example :

BASIC EXAMPLE

```
$Collection = 1,2,3,4,5  #variable holds array of integers
ForEach($Item in $Collection)
{
"Current Item =  $Item"
}

RESULT:

Current Item =  1
Current Item =  2
Current Item =  3
Current Item =  4
Current Item =  5
```

And there can many ways how you define a collection in a ForEach statement like the following examples :

- The collection is an array of items from 1 to 5 defined using a *Range operator*

```
ForEach($Item in 1..5)  #use range operator to define collection
{
    "Current Item = $Item"
}

RESULT:

Current Item = 1
Current Item = 2
Current Item = 3
Current Item = 4
Current Item = 5
```

- Define the collections as an *array of characters*

```
ForEach($Item in 'a','b','c','d','e')
{
    "Current Item = $Item"
}

RESULT:

Current Item = a
Current Item = b
Current Item = c
Current Item = d
Current Item = e
```

- Assign value of multiple variables to a single variable to make it a collection

```
$a = 5
$b = 6
$c = 7
$d = $a,$b,$c
Foreach ($i in $d)
{
 $i + 5
}

RESULT:
10
11
12
```

- Store outputs of PowerShell cmdlet or expression in a variable and pass that variable as a collection in foreach statement.

```
$Service =  Get-Service -Name "a*"
ForEach($s in $Service)
{
`    $s.Name

}

RESULTS:
AdobeARMservice
AJRouter
ALG
AppIDSvc
Appinfo
AppMgmt
AppReadiness
AppVClient
AppXSvc
AssignedAccessManagerSvc
AudioEndpointBuilder
Audiosrv
AxInstSV
```

- Run expression and commands directly in the Foreach statement to return a collection of items.In the following example, the Foreach statement steps through the list of items that is returned by the Get-ChildItem cmdlet.

```
ForEach($Item in Get-ChildItem C:\Test\*.txt)   #use command directly in foreach
loop
```

```
{
 "Current Item =  $Item"
}

RESULT:
Current Item =  C:\Test\notes.txt
Current Item =  C:\Test\test.txt
```

You can save the results generated by ForEach statement, all you have to do is add a variable just before the Foreach statement and all data would be stored in that variable which can be used to perform further operation based on your requirement.

Like to calculate the total CPU consumption by Processes with name chrome
A $CPU variable is added in front of Foreach statement to capture the result generated by the ForEach body

```
$CPU = Foreach($item in (Get-Process chrome*))  #store foreach output in a
variable
{
    $item.CPU
}
$totalCPU = ($CPU | measure -Sum).sum
"Total CPU: $totalCPU"

RESULT:
Total CPU: 751.25
```

If you want to do some filtration task on a collection of objects and perform some operation on filtered objects, then you can do like in the following example, print all services that starts with 'A' along with its status and start type. Moreover, in this example, you are not limited to running a single command in a statement list within foreach loop

```
$Service =  Get-Service
ForEach($S in $Service)
{
    If($s -like "A*")   #Use if block to do filter operation on current item in
loop
    {
        if($s.Status -eq "Stopped")
        {
            write-host "[$($s.Name)] : $($s.StartType) and Stopped"
        }
        else
```

```
        {
            Write-Host "[$($s.Name)] : $($s.StartType) and Running"
        }
    }
}



RESULTS:
[AdobeARMservice] : Automatic and Running
[AJRouter] : Manual and Stopped
[ALG] : Manual and Stopped
[AppIDSvc] : Manual and Stopped
[Appinfo] : Manual and Running
[AppMgmt] : Manual and Running
[AppReadiness] : Manual and Stopped
[AppVClient] : Disabled and Stopped
[AppXSvc] : Manual and Stopped
[AssignedAccessManagerSvc] : Manual and Stopped
[AudioEndpointBuilder] : Automatic and Running
[Audiosrv] : Automatic and Running
[AxInstSV] : Manual and Stopped
```

FOREACH-OBJECT CMDLET

ForEach-Object is a cmdlet (Not a loop) but just like a **Foreach statement**, it iterates through each object in a collection of an objects and perform operations on it.

A collection can be passed to the **-InputObject** parameter of the cmdlet or **can be piped**` to it. But output is same in both the cases. Let's understand this with following example:

BASIC EXAMPLE

```
# Input objects are passed to InputObject parameter
$process = Get-Process|Select-Object -First 10   #collection of objects

Foreach-Object -InputObject $process -Process {$_.Name}
                        OR
Foreach-Object -InputObject $process {$_.Name}

RESULT:
ApplicationFrameHost
armsvc
browser_broker
chrome
chrome
chrome
chrome
chrome
```

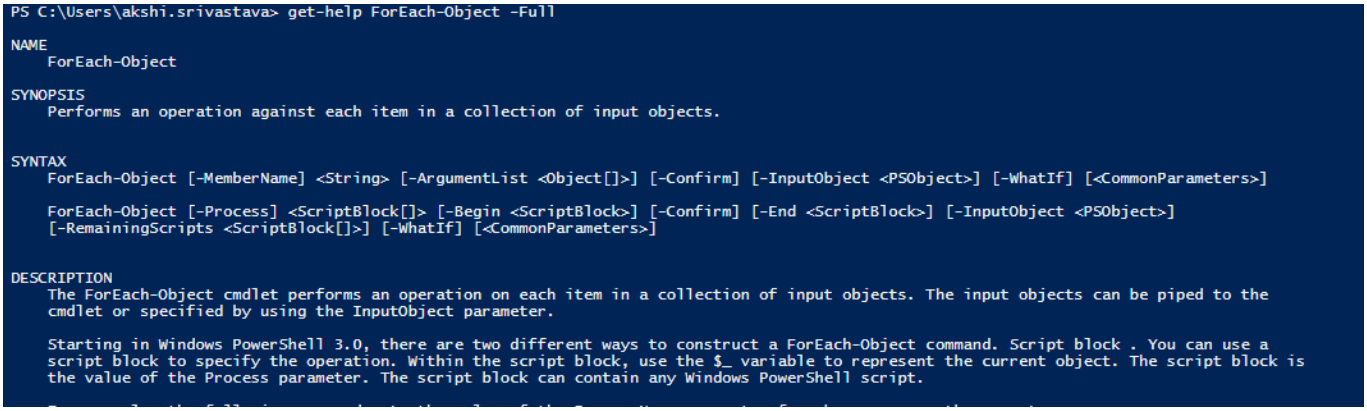
```
chrome
chrome


#Input objects are passed to cmdlet through pipeline
$process = Get-Process s*|Select-Object -First 10
$process|Foreach-Object -Process {$_.Name}

RESULT:
SAAZappr
SAAZDPMACTL
SAAZScheduler
SAAZServerPlus
SAAZWatchDog
sapisvr
SearchIndexer
SearchUI
SecurityHealthService
SecurityHealthSystray
```

In above example, curly braces '{}' is a scriptblock that is basically passed to the -Process parameter of this cmdlet, also known as **Process block** which iterates each item of the collection passed to Foreach-Object cmdlet and there is no need to specify -Process parameter before curly braces as it is the expected value at position '0'

Let's do Get-Help to this cmdlet and see few parameters and how they can be used with Foreach-Object cmdlet(Parameters are InputObject, Process and MemberName parameter)

```
Get-Help Foreach-Object -Full
```

This screenshot says about 2 parameter sets of Foreach-Object cmdlet and both set have one mandatory parameter i.e. MemberName and Process parameter respectively and one optional parameter InputObject in each set

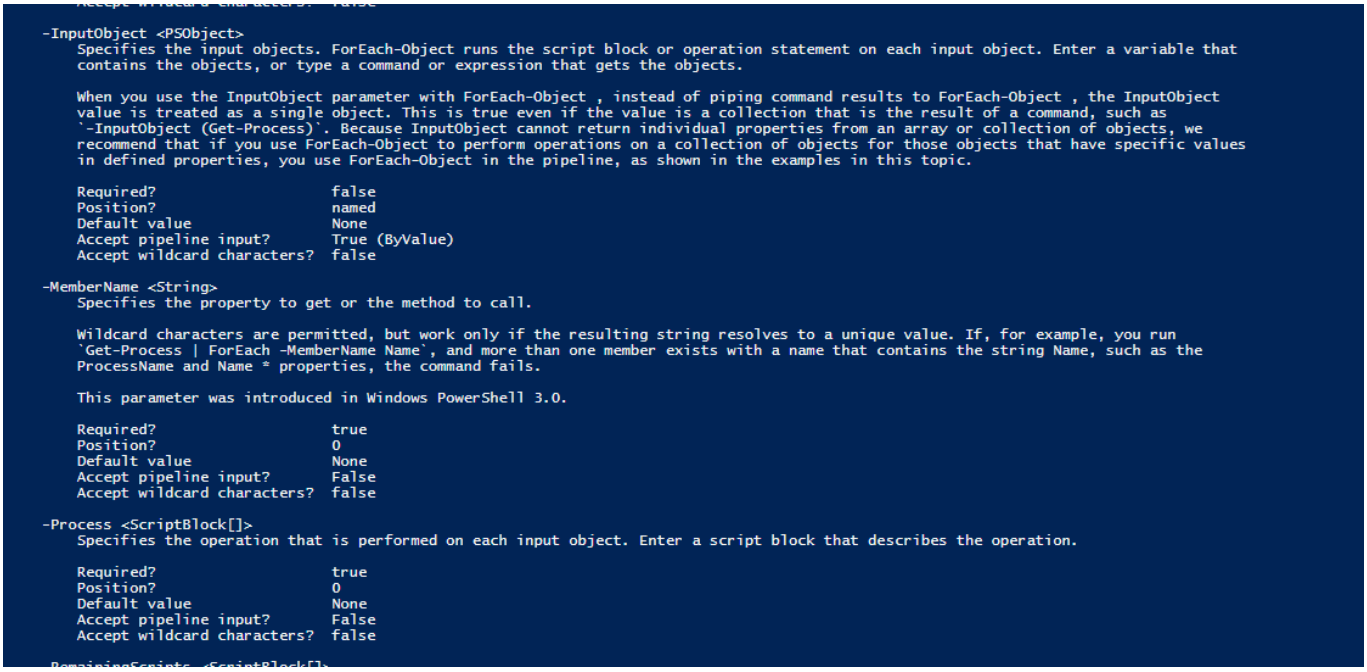


Below are the explanation of those 3 parameters:

InputObject - It accepts value via pipeline or can pass compelte object as a value to this parameter

Process - It accepts scriptblock as a value and it is positional parameter

MemberName - It either accepts property name or method of object as a value and it is also positional parameter



Basically, there are two different ways to construct a ForEach-Object command:

1. **SCRIPTBLOCK** : You can use a script block to specify the operation. Within the script block, use the **$_** or **$PSItem** variable to represent the current object. The script block is the value of the Process parameter which is the mandatory parameter when collection is passed either to InputObject parameter or via pipeline. The script block can contain any Windows PowerShell script.

```
1..5|Foreach-Object -Process {$_}  #scriptblock with pipeline input
Foreach-Object -InputObject (1..5) -Process {$_}  #scriptblock with InputObject
parameter

#Both above commands have same result
RESULT:
1
2
3
4
5
```

2. **OPERATION STATEMENT** : Operation statements were introduced in Windows PowerShell 3.0. You can also write an operation statement, which is much more like natural language. You can use the operation statement to specify a property name or call a method available for each object in a collection. Both

Property name and method name are passed as a value to 'MemberName' parameter. You can also use ArgumentList parameter of Foreach-Object cmdlet if you are invoking a method on an object and it requires some arguments to pass(multiple arguments can also be passed to ArgumentList parameter).

For example, the following command also gets the value of the ProcessName property of each process on the computer.

```
#MemberName parameter that accepts 'ProcessName' property of each object within a
collection as a value(input passed through pipeline)
Get-Process a* | ForEach-Object -MemberName ProcessName

RESULT:
ApplicationFrameHost
armsvc
```

In this example, we call a method Split on each string object that are passed through pipeline to Foreach-Object cmdlet and this method accepts dot(.) as an argument which would be passed as a value to ArgumentList parameter so that this method will split the each string object with dot(.)

```
#search if a split method can be used for the input collection by passing
collection to get-member cmdlet
"Microsoft.PowerShell.Core", "Microsoft.PowerShell.Host"|Get-Member
```

```
PS C:\Users\akshi.srivastava> "Microsoft.PowerShell.Core", "Microsoft.PowerShell.Host"|gm


   TypeName: System.String

Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone(), System.Object ICloneable.Clon
CompareTo        Method               int CompareTo(System.Object value), int CompareTo(st
Contains         Method               bool Contains(string value)
CopyTo           Method               void CopyTo(int sourceIndex, char[] destination, int
EndsWith         Method               bool EndsWith(string value), bool EndsWith(string va
Equals           Method               bool Equals(System.Object obj), bool Equals(string v
GetEnumerator    Method               System.CharEnumerator GetEnumerator(), System.Collec
GetHashCode      Method               int GetHashCode()
GetType          Method               type GetType()
GetTypeCode      Method               System.TypeCode GetTypeCode(), System.TypeCode IConv
IndexOf          Method               int IndexOf(char value), int IndexOf(char value, int
IndexOfAny       Method               int IndexOfAny(char[] anyOf), int IndexOfAny(char[]
Insert           Method               string Insert(int startIndex, string value)
IsNormalized     Method               bool IsNormalized(), bool IsNormalized(System.Text.N
LastIndexOf      Method               int LastIndexOf(char value), int LastIndexOf(char va
LastIndexOfAny   Method               int LastIndexOfAny(char[] anyOf), int LastIndexOfAny
Normalize        Method               string Normalize(), string Normalize(System.Text.Nor
PadLeft          Method               string PadLeft(int totalWidth), string PadLeft(int t
PadRight         Method               string PadRight(int totalWidth), string PadRight(int
Remove           Method               string Remove(int startIndex, int count), string Rem
Replace          Method               string Replace(char oldChar, char newChar), string R
Split            Method               string[] Split(Params char[] separator), string[] Sp
StartsWith       Method               bool StartsWith(string value), bool StartsWith(strin
Substring        Method               string Substring(int startIndex), string Substring(i
```

As you can at the end that we have Split method available to invoke on each string object. Hence, the command will be like:

```
#MemberName parameter that accepts 'Split' method of each string object within a
collection as a value(input passed through pipeline)
"Microsoft.PowerShell.Core", "Microsoft.PowerShell.Host" | ForEach-Object -
MemberName Split -ArgumentList "."

RESULT:
Microsoft
PowerShell
Core
Microsoft
PowerShell
Host
```

Foreach-Object does not supports Break and Continue branching statements. Let's understand the behaviour of Continue keyword in Foreach-Object cmdlet in following example.

It should print all combinations of 1 : a, 1: b, 1:d and similarly for other numbers. But here, just notice that after 1:b it is proceeding to 2:a without processing 1:d. That means continue statement inside a Foreach-Object will reach the for loop which is above the foreach-Object cmdlets and similarly.

```
for($i=1;$i -le 3; $i++){
    "a","b","c","d" | foreach-object {
        if($_ -eq "c") {
        continue
        }
    write-host "$i : $_"
    }
}

RESULT:
1 : a
1 : b
2 : a
2 : b
3 : a
3 : b
```

ForEach-Object has two aliases, ForEach and %,. The following three examples are identical in function, they have same output results:

```
Get-WMIObject Win32_LogicalDisk | ForEach-Object
{[math]::Round($_.FreeSpace/1GB,2)}
Get-WMIObject Win32_LogicalDisk | ForEach {[math]::Round($_.FreeSpace/1GB,2)}
Get-WMIObject Win32_LogicalDisk | % {[math]::Round($_.FreeSpace/1GB,2)}

RESULT:
197.7
195.21
```

Difference b/w Foreach and Foreach-Object

Foreach	**Foreach-Object**
It loads all of the items into a collection before processing them one at a time	Process only one item at a time through the pipeline
High memory utilisation	Less memory utilisation
If we are dealing with small collection of objects, it is always faster than Foreach-Object cmdlet	Comparitively slower than foreach loop
Cannot pass collection of objects to Foreach loop via pipeline	Collection of objects can either be passed through pipeline or to InputObject parameter
ForEach can also be piped to, because it's also an alias for ForEach-Object	But reverse can't be happened because, Foreach-Object is not a keyword and can't be used for loops
As pipeline is not being used by this, it won't allows you to stream the objects to another command via pipeline	We can pass the output to another command via pipeline
Foreach loop supports branching statements like Break and Continue	The Foreach-Object doesn’t support these branching statements

Performance measure of both Foreach loop and Foreach-Object cmdlet if we care dealing with small data

```
Measure-Command -Expression {Get-WMIObject Win32_LogicalDisk | ForEach-Object
{[math]::Round($_.FreeSpace/1GB,2)}}

RESULT:
Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 87
Ticks             : 872447
TotalDays         : 1.00977662037037E-06
```

```
TotalHours        : 2.42346388888889E-05
TotalMinutes      : 0.00145407833333333
TotalSeconds      : 0.0872447
TotalMilliseconds : 87.2447


Measure-command -Expression {$disks=Get-WMIObject Win32_LogicalDisk
foreach($disk in $disks)
{
   [math]::Round($disk.FreeSpace/1GB,2)
}
}

RESULT:
Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 34
Ticks             : 348787
TotalDays         : 4.03688657407407E-07
TotalHours        : 9.68852777777778E-06
TotalMinutes      : 0.000581311666666667
TotalSeconds      : 0.0348787
TotalMilliseconds : 34.8787
```

FOREACH METHOD

With the release of Windows PowerShell 4.0, a new **Magic Method** was introduced to support DSC that only works with `collection` known as **Foreach Method** that provides new syntax for accessing Foreach capabilities in Windows PowerShell. It allows you to rapidly loop through a collection of objects/array and execute a block of statement against each object in that collection.

Type @().Foreach() on PowerShell console where @() specifies collection and dot(.) operator is used to access Foreach() method on that collection and you will get the below result which tells you how we actually use this approach. This expression basically force an error which gives us a basic idea on how it can be used to iterate through the collection:

```
@().foreach()

RESULT:
Cannot find an overload for ".ForEach(expression [, arguments...])" and the
argument count: "0".
At line:1 char:1
+ @().foreach()
+ ~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodException
    + FullyQualifiedErrorId : MethodCountCouldNotFindBest```
```

The ForEach( ) method accepts a PowerShell ScriptBlock as its parameter that is executed once for each object in the collection. Within this ScriptBlock, the "current object" can be referred to using the *$_* or *$PSItem* automatic variables and you can see arguments paramater also that will be passed as an argument to the first paramater passed to a Foreach() method if required. We will understand the supported ways to invoke Foreach() method later in this article.

Hence, the PowerShell syntax for Foreach() method is:

Collection.Foreach ({scriptblock})

Input - A collection of objects e.g services, integers, server names etc.

Output - System.Collections.ObjectModel.Collection1[psobject] (Outputs are returned in a generic collection of this type.)

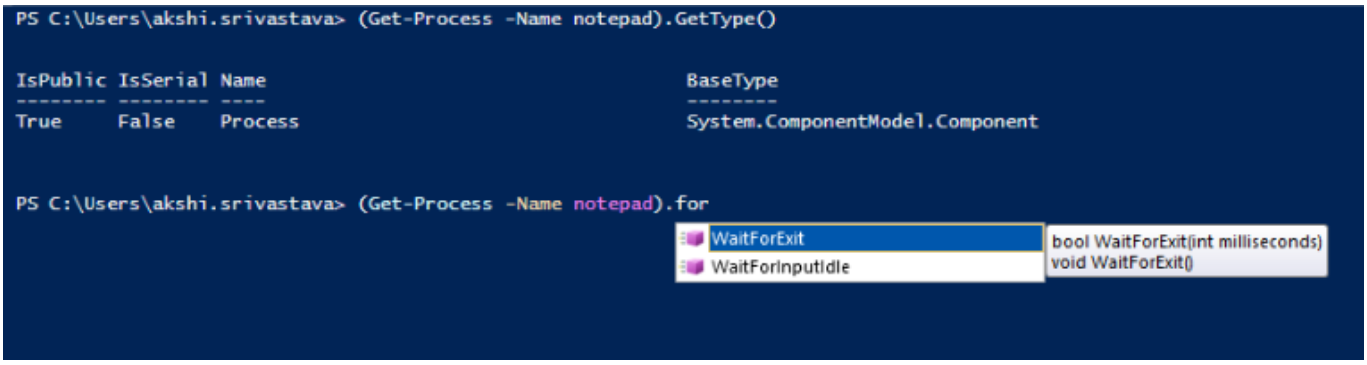
```
#Data type is array that can be passed as a collection to Foreach() method
(1..5).GetType

RESULT:
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array

#Using range operator to define the collection and passed it as an input to
Foreach method
(1..5).Foreach({$_})  #Using $_ to access current object
(1..5).Foreach({$PSItem})  #Using $PSItem to access current object

RESULT:   #both commands print same output
1
2
3
4
5

#Check type of output generated from Foreach method
((1..5).foreach({$_})).GetType()

RESULT:
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Collection`1                             System.Object
```

It's important to ensure that you are working with an array or collection, and not a single object. You will receive an error when trying to call the Foreach() method on a single object, because it only works on arrays/collection.

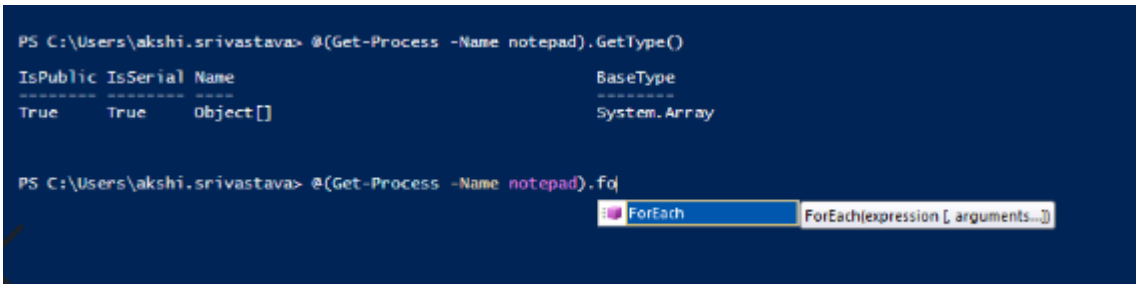
If it's possible for a command to return a single object, then as a best practice, you should wrap the command in @() to ensure that PowerShell treats the results as an array, even if only a single object is returned.

For example, a single process of notepad is running on your machine that means if you run get-process cmdlet on notepad the you will get single process object which is not a collection and hence you are not able to find Foreach() method by using dot(.) operator. But, if we want to use Foreach() method on this object then wrap this complete expression in @() to treat this as a collection/array.

```
#Get type of single notepad process
(Get-Process -Name notepad).GetType()

RESULT:
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Process
System.ComponentModel.Component
```




```
@(Get-Process -Name notepad).GetType()

RESULT
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```



In next example, we have multiple chrome processess which means it is an array of objects or a collection of chrome process objects, so, here it is not mandate to wrap the expression in @() if already know that we are passing a collection/array to Foreach() method but for best practice, use @() with Foreach() method

```
#Get type of multiple chrome processes
(Get-Process -Name chrome).GetType()

RESULT:
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

This method is called as **Magic Method* because they don't show up either in *Get-Member* output, even if you apply -Force and request -MemberType All or *.psobject.Methods*. They are private extension methods implemented on a private class. But when you use dot(.) operator with collection then you are able to see that method

```
1..5|Get-Member -Force
```

```
   TypeName: System.Int32

Name         MemberType   Definition
----         ----------   ----------
pstypenames  CodeProperty System.Collections.ObjectModel.Collection`1[[System.String, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyTo...
psadapted    MemberSet    psadapted {CompareTo, Equals, GetHashCode, ToString, GetTypeCode, GetType, ToBoolean, ToChar, ToSByte, ToByte, ToIn...
psbase       MemberSet    psbase {CompareTo, Equals, GetHashCode, ToString, GetTypeCode, GetType, ToBoolean, ToChar, ToSByte, ToByte, ToInt16...
psextended   MemberSet    psextended {}
psobject     MemberSet    psobject {Members, Properties, Methods, ImmediateBaseObject, BaseObject, TypeNames, get_Members, get_Properties, ge...
CompareTo    Method       int CompareTo(System.Object value), int CompareTo(int value), int IComparable.CompareTo(System.Object obj), int ICo...
Equals       Method       bool Equals(System.Object obj), bool Equals(int obj), bool IEquatable[int].Equals(int other)
GetHashCode  Method       int GetHashCode()
GetType      Method       type GetType()
GetTypeCode  Method       System.TypeCode GetTypeCode(), System.TypeCode IConvertible.GetTypeCode()
ToBoolean    Method       bool IConvertible.ToBoolean(System.IFormatProvider provider)
ToByte       Method       byte IConvertible.ToByte(System.IFormatProvider provider)
ToChar       Method       char IConvertible.ToChar(System.IFormatProvider provider)
ToDateTime   Method       datetime IConvertible.ToDateTime(System.IFormatProvider provider)
ToDecimal    Method       decimal IConvertible.ToDecimal(System.IFormatProvider provider)
ToDouble     Method       double IConvertible.ToDouble(System.IFormatProvider provider)
ToInt16      Method       int16 IConvertible.ToInt16(System.IFormatProvider provider)
ToInt32      Method       int IConvertible.ToInt32(System.IFormatProvider provider)
ToInt64      Method       long IConvertible.ToInt64(System.IFormatProvider provider)
ToSByte      Method       sbyte IConvertible.ToSByte(System.IFormatProvider provider)
ToSingle     Method       float IConvertible.ToSingle(System.IFormatProvider provider)
ToString     Method       string ToString(), string ToString(string format), string ToString(System.IFormatProvider provider), string ToStrin...
ToType       Method       System.Object IConvertible.ToType(type conversionType, System.IFormatProvider provider)
ToUInt16     Method       uint16 IConvertible.ToUInt16(System.IFormatProvider provider)
ToUInt32     Method       uint32 IConvertible.ToUInt32(System.IFormatProvider provider)
ToUInt64     Method       uint64 IConvertible.ToUInt64(System.IFormatProvider provider)
```

This new method works very similarly to the ForEach-Object commands that has existed in PowerShell since the beginning. They merely provide an alternate syntax for PowerShell developers who are hoping to use a fluent-style syntax in their code.

Seven supported ways to invoke this method :

1. collection.ForEach(scriptblock_expression)
2. collection.ForEach(scriptblock_expression, object[] arguments)
3. collection.ForEach(type convertToType)
4. collection.ForEach(string propertyName)
5. collection.ForEach(string propertyName, object[] newValue)
6. collection.ForEach(string methodName)

7. collection.ForEach(string methodName , object[] arguments)

Note that these are supported argument pairings, not different overloads available for the ForEach method. Using any argument pairings other than these may result in errors that do not clearly identify what the actual problem is.

ForEach(scriptblock expression)

If you pass a script block expression into the ForEach method, you are able to perform the same kind of tasks that you would do in a script block that you would use with the foreach statement or the ForEach-Object cmdlet.

```
#Get a set of services - collection
$services = Get-Service sa*
# Display the names and display names of all services in the collection  ****  1st
WAY  ****
$services.foreach({if($_.Status -eq "Running")
{$_.DisplayName}else{"$($_.DisplayName)[stopped]"}})

RESULT:
SAAZ RMM Agent Presence-PR
SAAZ RMM Agent Presence-SC[stopped]
SAAZDPMACTL
SAAZRemoteSupport[stopped]
SAAZScheduler
SAAZServerPlus
SAAZWatchDog
Security Accounts Manager
```

ForEach(scriptblock expression, object[] arguments)

Any arguments that you provide beyond the initial script block, will be used as arguments for the script block. This is just like how the -ArgumentList parameter works on the -Process parameter of ForEach-Object cmdlet. Arguments

```
#Get a set of services
$services = Get-Service sa*
# Select a property name to expand using a script block argument    ****  2nd WAY
****
$services.ForEach({Param($Name,$Status)
[pscustomobject]@{DisplayName=$_.$Name;Status=$_.$Status}},'DisplayName','Status')

RESULT:
DisplayName                 Status
-----------                 ------
SAAZ RMM Agent Presence-PR Running
```

```
SAAZ RMM Agent Presence-SC Stopped
SAAZDPMACTL                Running
SAAZRemoteSupport          Stopped
SAAZScheduler              Running
SAAZServerPlus             Running
SAAZWatchDog               Running
Security Accounts Manager  Running
```

ForEach(type convertToType)

You can also pass a data type into the ForEach method if you want to convert every item in a collection into another type. For example, imagine you have a collection of objects and you want to convert those objects into their string equivalent. Here is what that would look like with the ForEach method:

```
#the type of each object in the collection is System.Int32
1..5|gm
```

```
   TypeName: System.Int32

Name        MemberType   Definition
----        ----------   ----------
pstypenames CodeProperty System.Collections.ObjectModel.Collection`1[[System.String, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyTo...
psadapted   MemberSet    psadapted {CompareTo, Equals, GetHashCode, ToString, GetTypeCode, GetType, ToBoolean, ToChar, ToSByte, ToByte, ToIn...
psbase      MemberSet    psbase {CompareTo, Equals, GetHashCode, ToString, GetTypeCode, GetType, ToBoolean, ToChar, ToSByte, ToByte, ToInt16...
psextended  MemberSet    psextended {}
psobject    MemberSet    psobject {Members, Properties, Methods, ImmediateBaseObject, BaseObject, TypeNames, get_Members, get_Properties, ge...
CompareTo   Method       int CompareTo(System.Object value), int CompareTo(int value), int IComparable.CompareTo(System.Object obj), int ICo...
Equals      Method       bool Equals(System.Object obj), bool Equals(int obj), bool IEquatable[int].Equals(int other)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
GetTypeCode Method       System.TypeCode GetTypeCode(), System.TypeCode IConvertible.GetTypeCode()
ToBoolean   Method       bool IConvertible.ToBoolean(System.IFormatProvider provider)
ToByte      Method       byte IConvertible.ToByte(System.IFormatProvider provider)
ToChar      Method       char IConvertible.ToChar(System.IFormatProvider provider)
ToDateTime  Method       datetime IConvertible.ToDateTime(System.IFormatProvider provider)
ToDecimal   Method       decimal IConvertible.ToDecimal(System.IFormatProvider provider)
ToDouble    Method       double IConvertible.ToDouble(System.IFormatProvider provider)
ToInt16     Method       int16 IConvertible.ToInt16(System.IFormatProvider provider)
ToInt32     Method       int IConvertible.ToInt32(System.IFormatProvider provider)
ToInt64     Method       long IConvertible.ToInt64(System.IFormatProvider provider)
ToSByte     Method       sbyte IConvertible.ToSByte(System.IFormatProvider provider)
ToSingle    Method       float IConvertible.ToSingle(System.IFormatProvider provider)
ToString    Method       string ToString(), string ToString(string format), string ToString(System.IFormatProvider provider), string ToStrin...
ToType      Method       System.Object IConvertible.ToType(type conversionType, System.IFormatProvider provider)
ToUInt16    Method       uint16 IConvertible.ToUInt16(System.IFormatProvider provider)
ToUInt32    Method       uint32 IConvertible.ToUInt32(System.IFormatProvider provider)
ToUInt64    Method       uint64 IConvertible.ToUInt64(System.IFormatProvider provider)
```

```
#can perform addition on each object in the collection because it is of integer
type
(1..5).foreach({$_+2})

RESULT:
3
4
5
6
7

#change type of each object from System.Int32 to System.String
$result1=(1..5).foreach([string])
```

```
#the data type of each object in the collection is changes to System.String
$result1 | Get-Member
```

```
#check the data type of each object in the collection
$result1 | Get-Member


   TypeName: System.String

Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone(), System.Object ICloneable.Clone()
CompareTo        Method               int CompareTo(System.Object value), int CompareTo(string strB)
Contains         Method               bool Contains(string value)
CopyTo           Method               void CopyTo(int sourceIndex, char[] destination, int destinati
EndsWith         Method               bool EndsWith(string value), bool EndsWith(string value, Syste
Equals           Method               bool Equals(System.Object obj), bool Equals(string value), boo
GetEnumerator    Method               System.CharEnumerator GetEnumerator(), System.Collections.IEnu
GetHashCode      Method               int GetHashCode()
GetType          Method               type GetType()
GetTypeCode      Method               System.TypeCode GetTypeCode(), System.TypeCode IConvertible.Ge
IndexOf          Method               int IndexOf(char value), int IndexOf(char value, int startInde
IndexOfAny       Method               int IndexOfAny(char[] anyOf), int IndexOfAny(char[] anyOf, int
Insert           Method               string Insert(int startIndex, string value)
IsNormalized     Method               bool IsNormalized(), bool IsNormalized(System.Text.Normalizati
LastIndexOf      Method               int LastIndexOf(char value), int LastIndexOf(char value, int s
LastIndexOfAny   Method               int LastIndexOfAny(char[] anyOf), int LastIndexOfAny(char[] an
Normalize        Method               string Normalize(), string Normalize(System.Text.Normalization
PadLeft          Method               string PadLeft(int totalWidth), string PadLeft(int totalWidth,
PadRight         Method               string PadRight(int totalWidth), string PadRight(int totalWidt
Remove           Method               string Remove(int startIndex, int count), string Remove(int st
Replace          Method               string Replace(char oldChar, char newChar), string Replace(str
Split            Method               string[] Split(Params char[] separator), string[] Split(char[]
StartsWith       Method               bool StartsWith(string value), bool StartsWith(string value, S
Substring        Method               string Substring(int startIndex), string Substring(int startIn
```

```
#Can perform concatenation on each object in the collection after doing type
conversion to String
$result1.foreach({$_+2})

RESULT:
12
22
32
42
52
```

ForEach(string propertyName)

You can iterate through a collection of objects and return a values for a particular property of an object within that collection. In this case, it will only works against a single property name as a argument, except this will cause it to throw an error. To list more than one property, we have to use param statement just like we have seen in previous section (2nd supported way to define Foreach() method).

```
# Return the names of those services which starts from we
(Get-Service we*).foreach('Name')

RESULT:
```

```
WebClient
Wecsvc
WEPHOSTSVC
wercplsupport
WerSvc
```

ForEach(string propertyName, object[] newValue)

Not only we can retrieve a vaue of property of each objects within the collection, but can also set a value of property of each object within a collection. This is functionality that is not available in the other foreach's, unless you explicitly create the script block to do so. To set the property, you simply provide the property name and the value you want to use as an arguments in Foreach() method when setting that property like in below example. PowerShell will attempt to convert the new value you have provided as an argument to set value for the property into the appropriate type.

```
#Get DisplayName of service whose name starts with co
$service = (Get-Service co*|Select-Object -Property DisplayName)

RESULT:
DisplayName
-----------
COM+ System Application
ConsentUX_1e8402
CoreMessaging

# Now change the display names of every service to some new value
$service.foreach('DisplayName','Hello')
$service

RESULT:
DisplayName
-----------
Hello
Hello
Hello
```

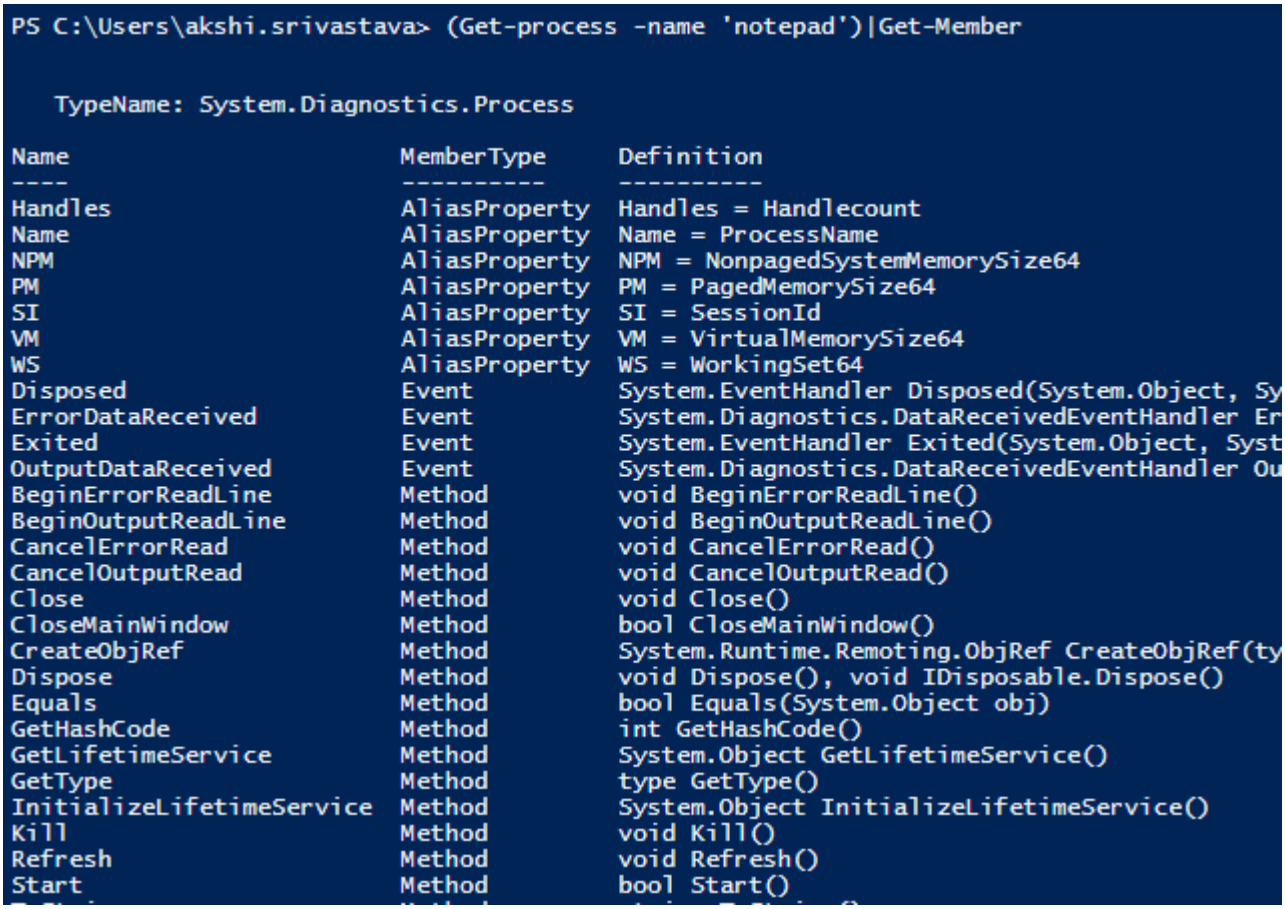
Note that the value of property of each collection's object is changed temporarily unless we redefine this variable or session is active. We can use that updated collection further to do other required operation.

ForEach(string methodName)

Method of an object in the collection can also be invoked. You just simply provide the method name as the argument to the foreach method and it will be invoked withour any arguments. Just make sure that if you are passing only method name in foreach() method, then the method does not accept any argument.

Here's an example showing how you could kill a bunch of processes running a specific program by using a kill method as an argument to Foreach()method

```
Start-Process -FilePath  Notepad.Exe
(Get-process -name 'notepad')|Get-Member
```



```
PS C:\Users\akshi.srivastava> (Get-process -name 'notepad')|Get-Member


   TypeName: System.Diagnostics.Process

Name                      MemberType     Definition
----                      ----------     ----------
Handles                   AliasProperty  Handles = Handlecount
Name                      AliasProperty  Name = ProcessName
NPM                       AliasProperty  NPM = NonpagedSystemMemorySize64
PM                        AliasProperty  PM = PagedMemorySize64
SI                        AliasProperty  SI = SessionId
VM                        AliasProperty  VM = VirtualMemorySize64
WS                        AliasProperty  WS = WorkingSet64
Disposed                  Event          System.EventHandler Disposed(System.Object, Sy
ErrorDataReceived         Event          System.Diagnostics.DataReceivedEventHandler Er
Exited                    Event          System.EventHandler Exited(System.Object, Syst
OutputDataReceived        Event          System.Diagnostics.DataReceivedEventHandler Ou
BeginErrorReadLine        Method         void BeginErrorReadLine()
BeginOutputReadLine       Method         void BeginOutputReadLine()
CancelErrorRead           Method         void CancelErrorRead()
CancelOutputRead          Method         void CancelOutputRead()
Close                     Method         void Close()
CloseMainWindow           Method         bool CloseMainWindow()
CreateObjRef              Method         System.Runtime.Remoting.ObjRef CreateObjRef(ty
Dispose                   Method         void Dispose(), void IDisposable.Dispose()
Equals                    Method         bool Equals(System.Object obj)
GetHashCode               Method         int GetHashCode()
GetLifetimeService        Method         System.Object GetLifetimeService()
GetType                   Method         type GetType()
InitializeLifetimeService Method         System.Object InitializeLifetimeService()
Kill                      Method         void Kill()
Refresh                   Method         void Refresh()
Start                     Method         bool Start()
```

```
# use 'kill' method to kill the process of notepad
(Get-process -name 'notepad').ForEach('Kill')
```

ForEach(string methodName, object[] arguments)

A method along with arguments can also be passed as a arguments to this foreach method. In below example, we get list of all commands which have computername paramater and then pass this collection as an input to foreach method along with 2 arguments - ResolveParameter(a method for an object) and ComputerName(an argument to this method)

```
$cmds = Get-Command -ParameterName ComputerName
$cmds|Get-Member  #get the method and properties of object in which you can see
ResolveParameter method which will be used in Foreach() method
```

```
PS C:\Users\akshi.srivastava> $cmds = Get-Command -ParameterName ComputerName

PS C:\Users\akshi.srivastava> $cmds|Get-Member


   TypeName: System.Management.Automation.CmdletInfo

Name                MemberType     Definition
----                ----------     ----------
Equals              Method         bool Equals(System.Object obj)
GetHashCode         Method         int GetHashCode()
GetType             Method         type GetType()
ResolveParameter    Method         System.Management.Automation.ParameterMetadata ResolveParameter(string name)
ToString            Method         string ToString()
CommandType         Property       System.Management.Automation.CommandTypes CommandType {get;}
DefaultParameterSet Property       string DefaultParameterSet {get;}
Definition          Property       string Definition {get;}
```

```
# Now show a table making sure the parameter names and aliases are consistent
$cmds.foreach('ResolveParameter','ComputerName') | Format-Table Name,Aliases

RESULT:
Name         Aliases
----         -------
ComputerName {}
ComputerName {Cn}
ComputerName {Cn}
ComputerName {cn}
ComputerName {}
ComputerName {Cn}
ComputerName {Cn}
ComputerName {CN, __Server, IPAddress}
ComputerName {Cn}
ComputerName {Cn}
ComputerName {Cn}
ComputerName {Cn}
ComputerName {CN}
```

Difference b/w Foreach-Object and Foreach Method

Foreach-Object	**Foreach Method**
Collection is passed either via pipeline or to InputObject parameter	Foreach method works on collection with the help of dot operator
It is a PowerShell cmdlet	It is a method introduced with PowerShell 4.0
This has parameters such as -Begin and -End for refining your script, and also -Confirm and -WhatIf for testing	This method has no parameters,it accepts certain types of arguments only