# POWERSHELL OBJECTS

...

# TLO KNOWLEDGE AND SKILLS

**Conditions:**

- Given a classroom, applicable references, and a practical exercise, the Cyber Mission Force student will demonstrate an understanding of PowerShell objects.

**Knowledge:**

- Identify PowerShell Objects.
- Understand the use of conditional statements.
- Identify how pipelines and one-liners.

**Skills:**

- Working knowledge of how to write conditional statements.
- Understanding of how to create loops and basic scripts.

# OBJECTIVES

- Identify Windows Powershell Objects
- Describe PowerShell Pipeline
- Describe Pipeline one-liner
- Describe pipeing PowerShell Cmdlets
- Iddentify Powershell Conditionals
- Describe Powershell If statements
- Identify Powershell Loops
  - Describe Powershell For Loops
  - Describe powershell Foreach Loops
- Describe Powershell Scripts
- Describe Powershell Execution Policies

# POWERSHELL OBJECTS

PowerShell is an object-oriented language and shell. This is a departure from the traditional shells like cmd and Bash.

These traditional shells focused on text aka strings and while still useful, are limited in their capabilities. Nearly everything in PowerShell is an object.

We can maximize Powershell’s potential by leveraging its object oriented structure to accomplish our goals.

# POWERSHELL PIPELINE

A pipeline is a series of commands connected by pipeline operators: |

Each pipeline operator sends the results of the preceding command to the next command.

The output of the first command can be sent for processing as input to the second command. And that output can be sent to yet another command.

The result is a complex command chain or pipeline that is composed of a series of simple commands.

# POWERSHELL PIPELINE

```
notepad.exe
Get-Process notepad | stop-process
```

In the above example, we started an instance of notepad.exe

We then use Get-Process with the name of "notepad" and PIPE that object into the cmdlet Stop-Process.

The notepad process then stops. We took the object from the left side of the pipeline and piped it as the input to the right side of the pipeline.

Simply put, the pipeline takes the output of an object and pipes it as the input of the next item in the pipe.

# ONE-LINER

A PowerShell one-liner is one continuous pipeline and not necessarily a command that's on one physical line. Not all commands that are on one physical line are one-liners.

Even though the following command is on multiple physical lines, it's a PowerShell one-liner:

```
Get-Service |
   Where-Object StartType -eq Automatic |
     Select-Object -Property Name
```

# ONE-LINER

Let’s break down what is happening in this one liner:

Step 1: The cmdlet Get-Service is ran and instead of the output being printed to the screen it is then piped into the next item in the pipe.

Step 2: The Where-Object cmdlet takes the previous item in the pipes output (in this case a Collection of services) as input and outputs each object that matches the Where-Object filter clause. The filter is looking for any service from the collection whose property "StartType" equals "Automatic".

Step 3: The Select-Object cmdlet takes the collection of objects from the previous pipe as input, it then selects **only** the objects property called "Name". Because it is the last pipe, it outputs it to the screen.

```
Get-Service |
   Where-Object StartType -eq Automatic |
     Select-Object -Property Name
```

# POWERSHELL OBJECTS

Objects have many different types of information associated with them.

In PowerShell, this information is sometimes called members.

An object member is a generic term that refers to all information associated with an object.

An object can have multiple members. But each member belongs to a single Object.

```
PS C:\Users\student\Downloads> get-process | get-member


   TypeName: System.Diagnostics.Process

Name                       MemberType     Definition
----                       ----------     ----------
Handles                    AliasProperty  Handles = Handlecount
Name                       AliasProperty  Name = ProcessName
NPM                        AliasProperty  NPM = NonpagedSystemMemoryS...
PM                         AliasProperty  PM = PagedMemorySize64
SI                         AliasProperty  SI = SessionId
VM                         AliasProperty  VM = VirtualMemorySize64
WS                         AliasProperty  WS = WorkingSet64
Disposed                   Event          System.EventHandler Dispose...
ErrorDataReceived          Event          System.Diagnostics.DataRece...
Exited                     Event          System.EventHandler Exited(...
OutputDataReceived         Event          System.Diagnostics.DataRece...
BeginErrorReadLine         Method         void BeginErrorReadLine()
BeginOutputReadLine        Method         void BeginOutputReadLine()
CancelErrorRead            Method         void CancelErrorRead()
CancelOutputRead           Method         void CancelOutputRead()
Close                      Method         void Close()
CloseMainWindow            Method         bool CloseMainWindow()
```

# POWERSHELL OBJECTS

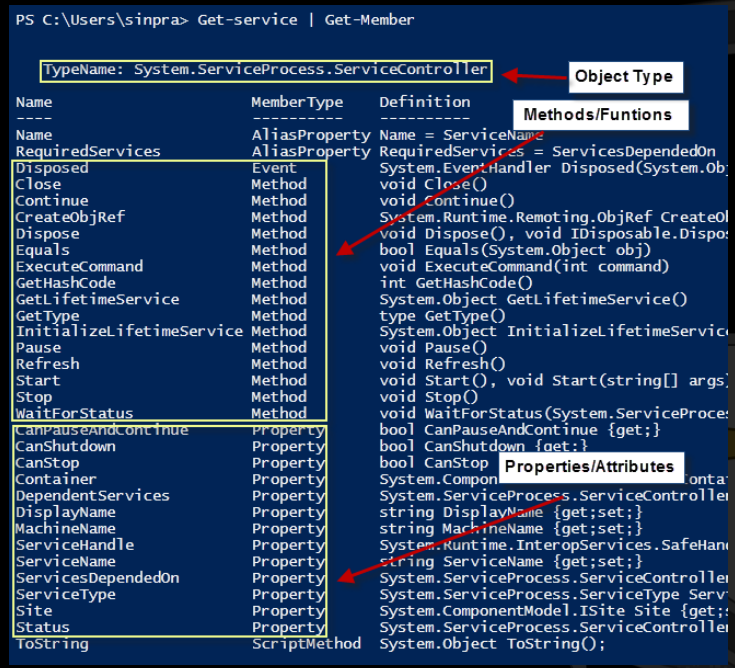
To discover information about an object, you can use the Get-Member cmdlet.

The Get-Member cmdlet is a handy command that allows you find available properties, methods, and so on for any object in PowerShell.

For example: If we take the cmdlet Get-service and pipe it into Get-Member:

Get-service | Get-Member

We can see that the Get-service cmdlet returns a System.serviceProcess.t. This object has members that belong to that object.



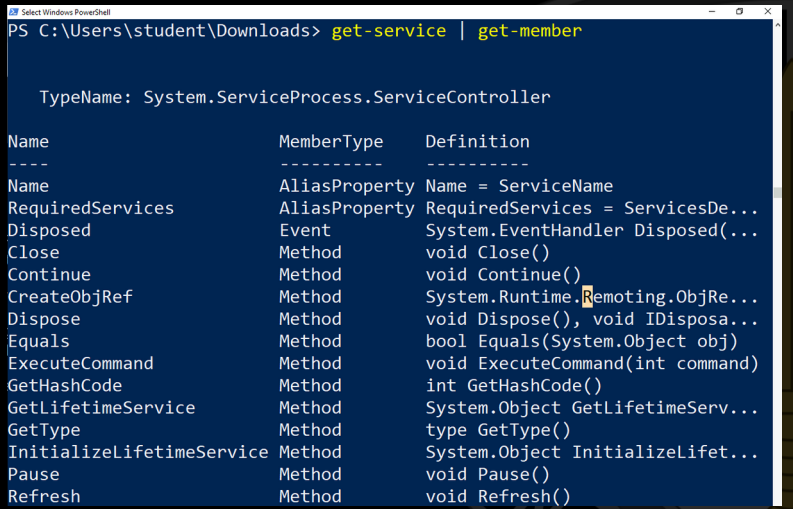
# POWERSHELL OBJECTS

We can Pause, Start, and Stop services using the Methods: Pause, Stop and Start.

These are actions taken on an object.

We can get information about each service by looking at the Properties of that object.

Properties such as DisplayName, ServiceName and Status etc.



# POWERSHELL OBJECTS

The Method member are Methods that can be called. Methods are actions that can be taken on an object. In the human example, a Method of the Human Object would be Walk, Run, Breath. This are actions that immediately affect that object that calls them.

The Property members are attributes that describe an object. An object can have many different properties attached to it representing various attributes.

If we want to search for these properties we can use the Where-Object.

# POWERSHELL CONDITIONALS

All procedural programming languages require constructs that allow them to execute instructions that are dependent on certain conditions.

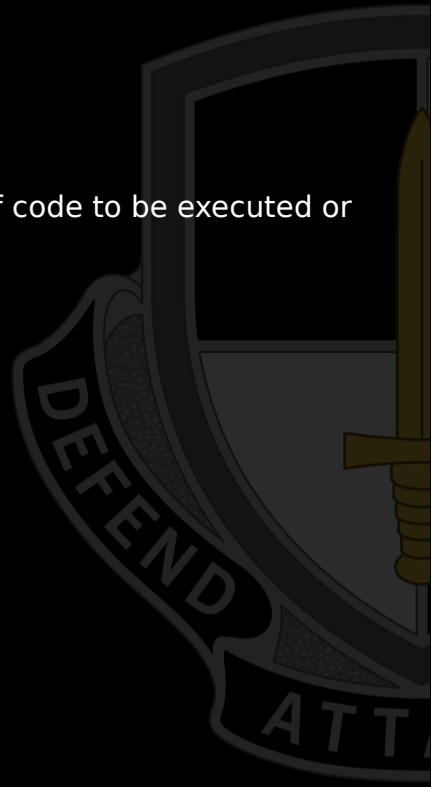
Procedural programming is a programming paradigm, derived from imperative programming, based on the concept of the procedure call.

Procedures simply contain a series of computational steps to be carried out.

# IF STATEMENT

Because Powershell is procedural in nature, it has built in conditionals that allow certain blocks of code to be executed or ignored based on a specific condition.

These conditions in Powershell are determined using “If” statements.

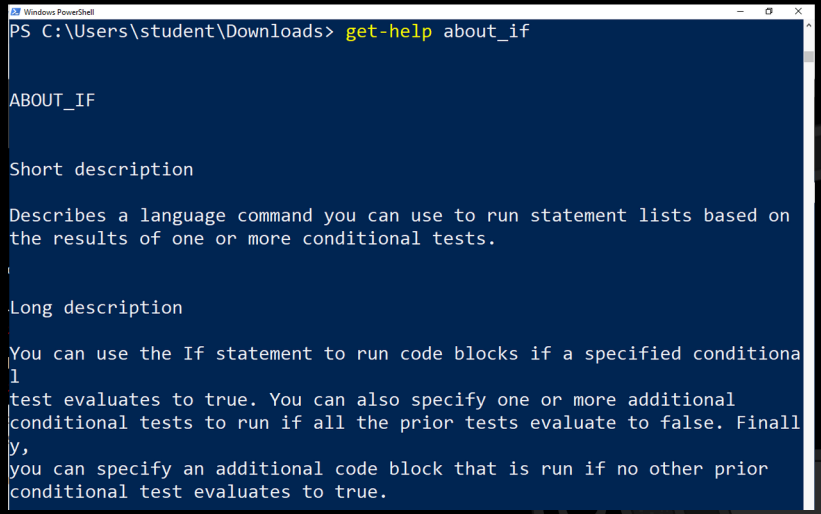


# IF STATEMENT

The first thing the if statement does is evaluate the expression in parentheses.

If it evaluates to $true, then it executes the scriptblock in the braces.

If the value was $false, then it would skip over that scriptblock.




```
$condition = $true
if ( $condition )
{
    Write-Output "The condition was true"
}
```

```
$x = 1
if ( $x -eq 1 )
{
    Write-Output "The condition was true"
}
```

# ELSE STATEMENT

The else statement does not accept any condition.

The statement list in this statement contains the code to run if all the prior conditions tested are false.

Else will always resolve to $True if all conditions prior to it are False.

Think of the else as a last resort condition.

```
$condition = $false
if ( $condition )
{
    Write-Output "The condition was true"
}
else
{
    Write-Output "The condition was false"
}
```

# ELSEIF STATEMENT

The elseif statement is where you add conditions.

You can add multiple ElseIf statements when you multiple conditions.

PowerShell will evaluate each of these conditions sequentially.

```
$condition = $false
if ( $condition )
{
    Write-Output "The condition was true"
}
elseif (!$condition)
{
    Write-Output "The condition was false"
}
else
{
    Write-Output "I dont know how you got here."
}
```

# COMPARISON OPERATORS

The most common use of the if statement is comparing two items with each other.

PowerShell has special operators for different comparison scenarios.

When you use a comparison operator, the value on the left-hand side is compared to the value on the right-hand side.

# NOT -! OPERATOR

not - !

The not operand simply inverts the boolean value of a condition or boolean object.

A false statement becomes true and a true statement becomes false, just like in Computer Organization and Architecture.

```
$condition = $false
if ( !$condition )
{
    Write-Output "The condition was true"
}
Write-Output (!$condition)
```

# OTHER OPERATORS

eq - Equal

```
$value = 5
if ( 5 -eq $value )
{
    Write-Output "The value is $value"
}
```

ne - Not Equal

```
$value = 4
if ( 5 -ne $value )
{
    Write-Output "The value $value is not 5"
}
```

gt - Greater Than

```
$value = 4
if ( 5 -gt $value )
{
    Write-Output "The value 5 is greater than $value"
}
```

ge - Greater Than or Equal

```
$value = 5
if ( 5 -ge $value )
{
    Write-Output "The value 5 is greater than or equal to $value"
}
```

# OTHER OPERATORS

lt - Less Than

```
$value = 6
if ( 5 -lt $value )
{
    Write-Output "The value 5 is less than $value"
}
```

le - Less Than or Equal

```
$value = 5
if ( 5 -le $value )
{
    Write-Output "The value 5 is less than or equal to $value"
}
```

is - Is the same type

```
$value = "test"
if ( $value -is [string] )
{
    Write-Output "The value $value is a string"
}
```

# FOR LOOP

Init - The Init placeholder represents one or more commands that are run before the loop begins.

Condition - The Condition placeholder represents the portion of the For statement that resolves to a $true or $false Boolean value. PowerShell evaluates the condition each time the For loop runs. If the statement is $true, the commands in the command block run. The loop is repeated until the condition becomes $false.

Repeat - The Repeat placeholder represents one or more commands, separated by commas, that are executed each time the loop repeats.

Statement - The Statement list placeholder represents a set of one or more commands that are run each time the loop is entered or repeated. The contents of the Statement list are surrounded by braces.

```
for (<Init>; <Condition>; <Repeat>)
{
    <Statement list>
}
```

```
for ($i;$i -lt 15; $i++)
{
    Write-Output "Loop Number: $i"
}
$i = 0
```

# FOR EACH LOOP

A foreach loop reads a set of objects (iterates) and completes when it’s finished with the last one.

The simplest and most typical type of collection to traverse is an array. Within a foreach loop, it is common to run one or more commands against each item in an array.

The difference between a For loop and Foreach is that a for loop is typically looped until a condition is true.

It will loop forever until that condition is true.

In a foreach loop a set amount of loops are determined based on the size of a collection.

The $<item> is created locally within the loop and the variable is name by the user.

```
foreach ($<item> in $<collection>)
{
    <statement list>
}
```

# POWERSHELL SCRIPTS

A script is a plain text file that contains one or more PowerShell commands. PowerShell scripts have a .ps1 file extension.

Running a script is a lot like running a cmdlet. You type the path and file name of the script and use parameters to submit data and set options. You can run scripts on your computer or in a remote session on a different computer.

Writing a script saves a command for later use and makes it easy to share with others. Most importantly, it lets you run the commands simply by typing the script path and the filename. Scripts can be as simple as a single command in a file or as extensive as a complex program.

Scripts have additional features, such as the #Requires special comment, the use of parameters, support for data sections, and digital signing for security. You can also write Help topics for scripts and for any functions in the script.

# EXECUTION POLICY

Before you can run a script on Windows, you need to change the default PowerShell execution policy. Execution policy does not apply to PowerShell running on non-Windows platforms.

In order to change the execution policy we will use the built-in powershell cmdlet Set-ExecutionPolicy.

# EXECUTION POLICY

- `AllSigned`
  - Scripts can run.
  - Requires that all scripts and configuration files be signed by a trusted publisher, including scripts that you write on the local computer.
  - Prompts you before running scripts from publishers that you haven't yet classified as trusted or untrusted.
  - Risks running signed, but malicious, scripts.
- `Bypass`
  - Nothing is blocked and there are no warnings or prompts.
  - This execution policy is designed for configurations in which a PowerShell script is built in to a larger application or for configurations in which PowerShell is the foundation for a program that has its own security model.
- `Default`
  - Sets the default execution policy.
  - Restricted for Windows clients.
  - RemoteSigned for Windows servers.
- `RemoteSigned`
  - The default execution policy for Windows server computers.
  - Scripts can run.
  - Requires a digital signature from a trusted publisher on scripts and configuration files that are downloaded from the internet which includes email and instant messaging programs.
  - Doesn't require digital signatures on scripts that are written on the local computer and not downloaded from the internet.
  - Runs scripts that are downloaded from the internet and not signed, if the scripts are unblocked, such as by using the Unblock-File cmdlet.
  - Risks running unsigned scripts from sources other than the internet and signed scripts that could be malicious.

# EXECUTION POLICY

- `Restricted`
  - The default execution policy for Windows client computers.
  - Permits individual commands, but does not allow scripts.
  - Prevents running of all script files, including formatting and configuration files (.ps1xml), module script files (.psm1), and PowerShell profiles (.ps1).
- `Undefined`
  - There is no execution policy set in the current scope.
  - If the execution policy in all scopes is Undefined, the effective execution policy is Restricted for Windows clients and RemoteSigned for Windows Server.
- `Unrestricted`
  - The default execution policy for non-Windows computers and cannot be changed.
  - Unsigned scripts can run. There is a risk of running malicious scripts.
  - Warns the user before running scripts and configuration files that are not from the local intranet zone.

You can find all the descriptions and differences by checking out the help page for about_Execution_policies