

```
PS> $title = "POWERSHELL"
```

```
PS> $title[0..9]
```

P

O

W

E

R

S

H

E

L

L

```
PS> Get-Instructors
```

Roque Sosa

Eduardo Garcia

Brief Contents

- 1 ■ What is Powershell?
- 2 ■ Basic Types & Variables
- 3 ■ Collections
- 4 ■ Operators
- 5 ■ Flow control
- 6 ■ Cmdlets & the Pipeline
- 7 ■ Integrated Scripting Environment (ISE)

What is Powershell?



Introduction

Powershell is Microsoft's response for the dominance on terminal-based tasks of Linux, fixing most issues that come with using and learning bash. In contrast with Bash, Powershell is easy to learn and read.

In-Depth

The technical definition of this technology is *"Powershell is a command-line shell, scripting environment and an automation engine that works using different versions of the .NET Framework"*. It's designed to be easily used, by implementing commands using plain English without erasing the possibility for "techies" to use more complicated implementations (e.g aliases and C# code).

Regardless of the path you opt to use, you can achieve the same results.

Powershell's aim is to replace the command prompt in the Windows environment, some say it already replaced it, giving the Windows user, now Mac and Linux too, the power and control over their OS.

Why is Powershell worth learning? And by Whom?

Powershell was designed for SysAdmins and DEVs on mind but anyone looking to make quick and easy automation should look towards Powershell, it is not the only option but, it is easier to learn and jump right into the action.

Let's see a couple of examples on its simplicity

- **Hello World**

- o Bash

```
Ras@ComputerName ~  
$ echo "Hello World"  
Hello World
```

- o Visual Basic

```
Imports System  
Imports System.Threading  
  
Module Program  
    Sub Main(args As String())  
        Console.WriteLine("Hello World!")  
        'As this is a console compiled project you need to  
stop the program to see the output  
        Thread.Sleep(1000)  
    End Sub  
End Module
```

- o Powershell

```
PS> "Hello World"  
Hello World
```

- **Visualize and filter data**

Let's say we have this set of objects containing certain information and we want to filter this information. There are ways to do this directly in the command prompt, but for a beginner, a UI is more friendly. So let's explore how we would go into creating a quick UI.

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	--	--	-----
480	22	8392	15892	126.58	4272	2	AdobeCollabSync
280	15	4644	9520	0.53	39808	2	AdobeCollabSync
534	30	30128	17876	25.58	7000	2	ApplicationFrameHost
166	9	2436	1020	0.17	9844	2	AppVShNotify
157	8	1928	1084		29432	0	AppVShNotify
321	16	3452	2288		5060	0	armsvc
121	9	1676	1356		3816	0	AsLdrSrv
181	15	2876	4364	0.56	12632	2	ATKOSD2
616	17	68732	17092	125.00	55084	0	audiodg
446	25	4888	6852	247.20	18440	2	AuthManSvr
431	24	6868	8988	1.64	56560	2	avgnt
1666	62	241264	52260		3396	0	avguard
1396	141	90832	5272		6080	0	Avira.ServiceHost
1041	175	87528	27224	1,462.20	1684	2	Avira.Systray
144	9	2028	6184		50108	0	avshadow
154	7	5652	6756	0.03	55980	2	bash
508	101	297068	264860	231.03	1988	2	chrome
321	11	2272	1952	1.69	6816	2	chrome
561	100	271980	186420	665.55	8032	2	chrome
335	90	182972	112084	105.25	11608	2	chrome
4940	43	367996	188508	6,878.16	15652	2	chrome
329	39	54832	63904	6.16	16636	2	chrome
296	32	59940	46636	44.36	17996	2	chrome
142	11	2028	1804	0.14	19708	2	chrome
553	34	49320	46232	2,316.02	24796	2	chrome
288	31	63124	32296	44.66	26484	2	chrome
292	75	130252	114308	362.67	31092	2	chrome
329	36	62244	55604	183.47	31316	2	chrome
287	29	52856	30292	17.20	31812	2	chrome
3600	94	377220	238528	13,809.50	341		

- Bash

```
Ras@ComputerName ~  
$ #It doesn't exist
```

- Visual Basic

Imports System

Module Program

Sub Main(args As String())

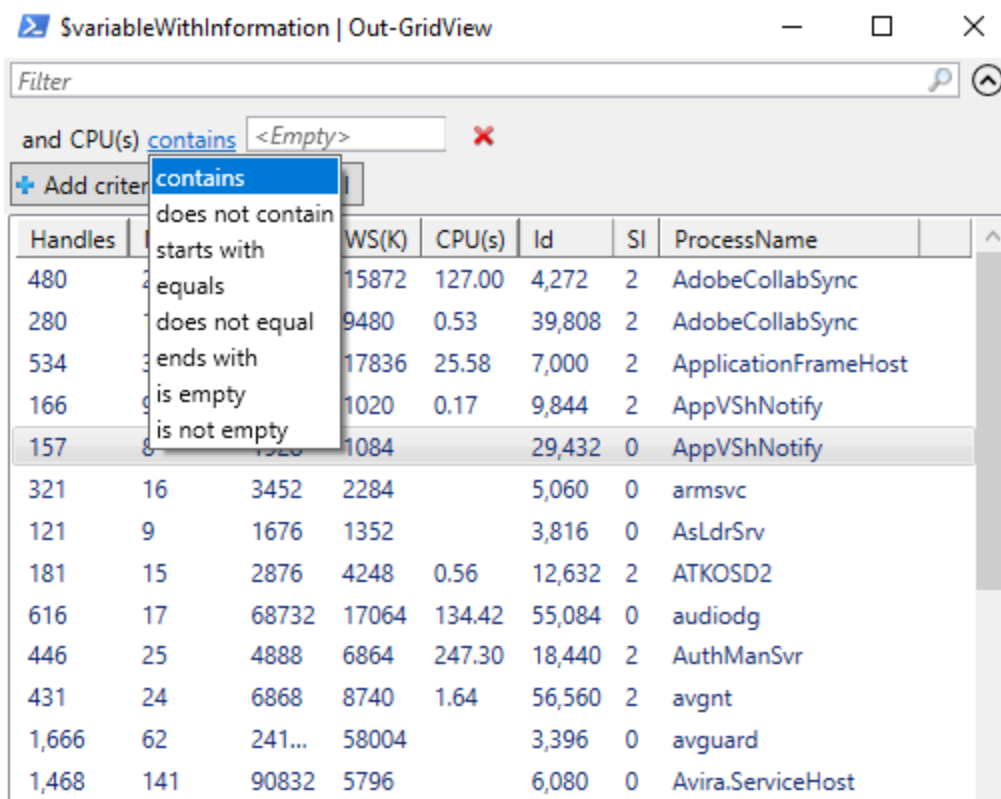
'It can be done but it'll be too complicated to take the time to make it just as an example

End Sub

End Module

- Powershell

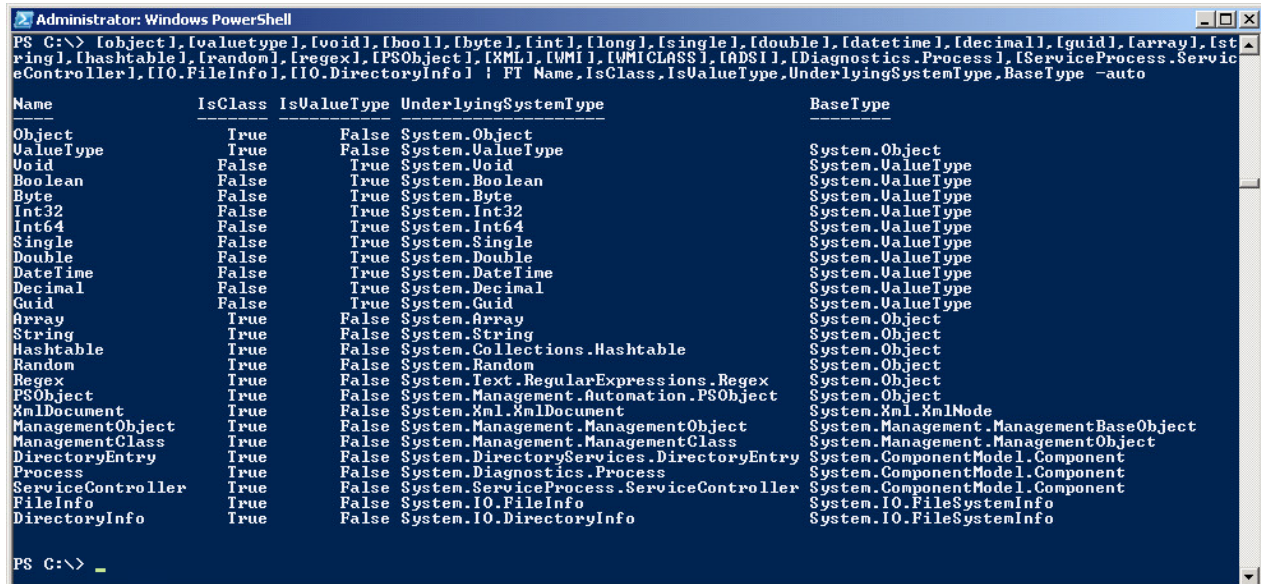
PS> \$variableWithInformation | Out-GridView



And just like this, there are several ways for beginners to start with Powershell. To be completely fair, as you begin to learn more and more, the use of this UI based tools/Cmdlets will be less productive but it is very useful in the beginning, take advantage of it while it lasts.

Well, I hope to have convinced you that Powershell is worth learning, without further ado, let's start with Powershell.

Basic Types & Variables



```
PS C:\> [object], [valuetype], [void], [bool], [byte], [int], [long], [single], [double], [datetime], [decimal], [guid], [array], [string], [hashtable], [random], [regex], [PSObject], [XML], [WMI], [WMICLASS], [ADSI], [Diagnostics.Process], [ServiceProcess.ServiceController], [IO.FileInfo], [IO.DirectoryInfo] | FT Name, IsClass, IsValueType, UnderlyingSystemType, BaseType -auto
```

Name	IsClass	IsValueType	UnderlyingSystemType	BaseType
Object	True	False	System.Object	System.Object
ValueType	False	True	System.ValueType	System.ValueType
Void	False	True	System.Void	System.ValueType
Boolean	False	True	System.Boolean	System.ValueType
Byte	False	True	System.Byte	System.ValueType
Int32	False	True	System.Int32	System.ValueType
Int64	False	True	System.Int64	System.ValueType
Single	False	True	System.Single	System.ValueType
Double	False	True	System.Double	System.ValueType
DateTime	False	True	System.DateTime	System.ValueType
Decimal	False	True	System.Decimal	System.ValueType
Guid	False	True	System.Guid	System.ValueType
Array	True	False	System.Array	System.Object
String	True	False	System.String	System.Object
Hashtable	True	False	System.Collections.Hashtable	System.Object
Random	True	False	System.Random	System.Object
Regex	True	False	System.Text.RegularExpressions.Regex	System.Object
PSObject	True	False	System.Management.Automation.PSObject	System.Object
XmlDocument	True	False	System.Xml.XmlDocument	System.Xml.XmlNode
ManagementObject	True	False	System.Management.ManagementObject	System.Management.ManagementBaseObject
ManagementClass	True	False	System.Management.ManagementClass	System.Management.ManagementObject
DirectoryEntry	True	False	System.DirectoryServices.DirectoryEntry	System.ComponentModel.Component
Process	True	False	System.Diagnostics.Process	System.ComponentModel.Component
ServiceController	True	False	System.ServiceProcess.ServiceController	System.ComponentModel.Component
FileInfo	True	False	System.IO.FileInfo	System.IO.FileSystemInfo
DirectoryInfo	True	False	System.IO.DirectoryInfo	System.IO.FileSystemInfo

Introduction

In Powershell, like in all other programming languages, data can be stored in variables for ease of use.

In-Depth

In Programming, **variables** are defined as values that can change depending on the conditions and information passed on to the script.

Let's say we have the variable \$color whose value is the word "Red". Now let's see what happens when we input \$color in the PowerShell console:

```
PS> $color  
Red
```

Instead of printing what we literally wrote we are given the variable's value "Red".

To use a Variable in Powershell, first it must be defined. A variable always starts with the dollar sign (\$) followed by any combination of letters and numbers. It's common practice to always start a variable with a lowercase letter

Remember! Try to use easy to read variable names, the best script is the one that's easily read without much digging into.

After defining the variable's name we equal it to the desired value. This means that if we want \$color to be "blue" we simply write down (this is better explained in the Operators chapter):

```
PS> $color = "blue"
```

And that's it! The variable is set and under the current session it will remain as that value unless changed! Changing a value is as easy and simple as equalling to another value, like this:

```
PS> $color = 42
```

Now \$color is no longer the word "blue", but instead the number 42, as easy as that.

Variables can be one of many types. We will proceed to see some of the following.

Integers

An Integer is, as its name suggests, a ***whole number (without decimals)***. It's the most basic type of number value. Defining it is done by just equalling the desired value to it's variable name:

```
PS> $employees = 50  
PS> $age = 26
```

Do keep in mind that if the value is wrapped in quotes or single quotes it stops being an Integer and it's a String instead! We'll see more about that in a second.

Doubles

If Integers are whole numbers then Doubles are floating point numbers instead. Meaning those that possess decimal places. Defined in the same way that Integers are defined:

```
PS> $thisIsADouble = 1.53
```

Strings

A String is a **sequence of characters** (an array of [char]), what we commonly understand as a word, sentence or even an entire book. They can be comprised of letters, numbers, symbols and punctuation marks. Declaring it is done by wrapping the string itself in quotes, either double (") or simple ('):

```
PS> $newString = "Powershell is Awesome!"
```

You may ask yourself, what is the difference between using either type of quotes then? Inside Strings you can write down other variables and it will take their values if you use double quotes, if using simple quotes it will take instead the written name, for example:

```
PS> $variable = "Powershell"
```

```
PS> "$variable is Awesome!"  
Powershell is Awesome!
```

```
PS> '$variable is Awesome!'  
$variable is Awesome!
```

Double quotes can also be told not to solve for the values of the variables by adding to the name of the variables the character (`):

```
PS> $variable = "Powershell"  
PS> "`$variable is Awesome!"  
$variable is Awesome!
```

Try mixing up different quotations and you'll get diverse results!.

Booleans

Booleans, the odd ones in the bunch, while the other data types are easily translated to a real life equivalent, booleans are something a bit different. A Boolean is **a value that is either true or false**. It's value equalling to a positive or a negative statement.

```
PS> $isPowershellCool = $true
PS> $flatEarth = $false
```

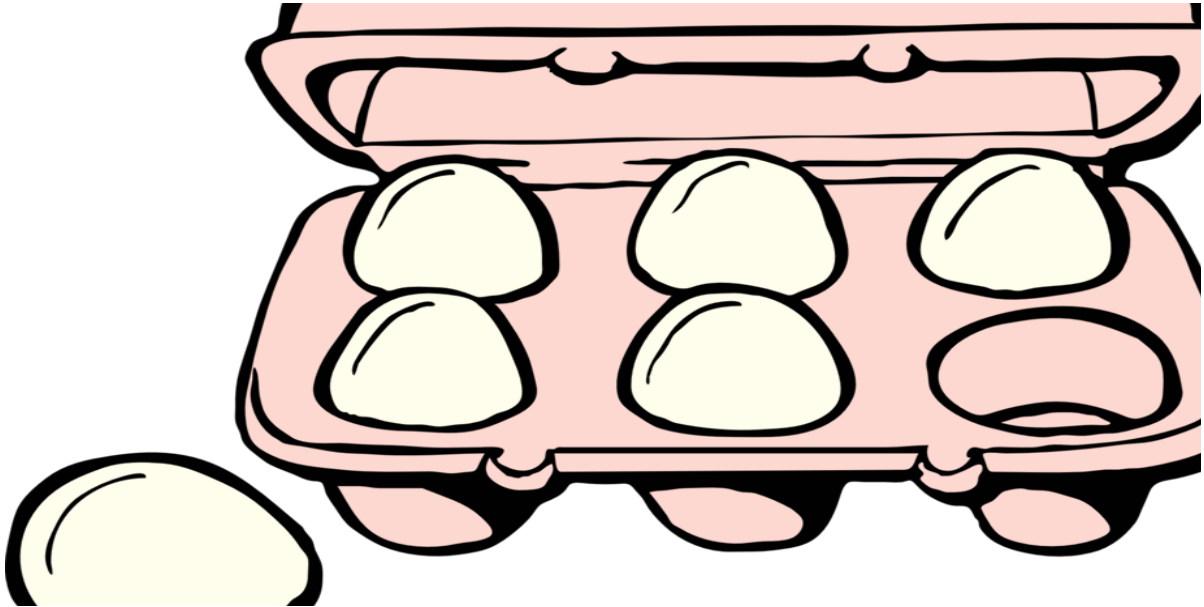
Zero Values

Each data type has its corresponding zero value, the value it possesses when not initialized. This would be the default value for each data type, some of them are:

- "" as an empty String
- An Integer being 0 while a Double being 0.0
- Booleans directly default to \$false

In case a variable hasn't been defined, then it's value is \$null and it's type "undefined".

Collections



Introduction

A Collection is quite simply **an object that stores multiple objects** in an easy to access, easy to handle manner. They are used to store, retrieve and manipulate data.

Arrays

An Array in Powershell is a **fixed size** collection of data items, which don't necessarily have to be of the same data type. They are indexed in a fixed position. To define an array, simply name it as a regular variable (for example `$animals`) and then proceed to wrap its values inside a `@` sign and parentheses.

```
PS> $animals = @("dog", "cat", "horse")
```

As previously mentioned they can also contain different data types:

```
PS> $someValues = @("something", 55, $true)
```

Arrays have a fixed length, accessing a value inside an array is done by its **index number**. Index numbers always start at 0, meaning **0 belongs to the first value in an array**.

```
PS> $animals[0]  
dog
```

```
PS> $animals[1]  
Cat
```

```
PS> $animals[2]  
Horse
```

Values inside an array can also be modified directly:

```
PS> $animals[2] = "donkey"  
PS> $animals[2]  
donkey
```

Arrays are easy to make and easy to use.

ArrayLists

An ArrayList is a class that combines the **storage architecture and the variability** in size of a List with the **ease of use** of an Array. Just like with normal arrays it's values can be accessed by index number. An ArrayList is defined by declaring it as a new Object.

```
PS>$arrayList = New-Object System.Collections.ArrayList  
or  
PS>$arrayList = [System.Collections.ArrayList]::new()
```

New values can be added to it by using its method "Add" (*it prints to screen the current index of addition add " | Out-Null" to avoid it*)

```
PS> $countries.Add("USA")  
0
```

By doing this its first index (remember, **it's 0!**) becomes "USA". Future additions are then added to the end of the list

```
PS> $countries.Add("Argentina")  
1
```

HashTables

A Hash Table is another kind of collection, a very useful one that used correctly can make accessing an element inside it much more intuitive.

While arrays and arraylists use indexes to locate their values, hashtables instead use a **pair of "key" and "value"** to access each element. When you need to access a desired value you need only provide the respective key instead of an index number.

The easiest way to see this is by looking at the table like an object, each key being a property with a determined value. This type of collection is defined by using the @ sign followed by curly braces. Then each key is equalled to a value. Imagine we have a car, with color, model, manufacturer and number of doors:

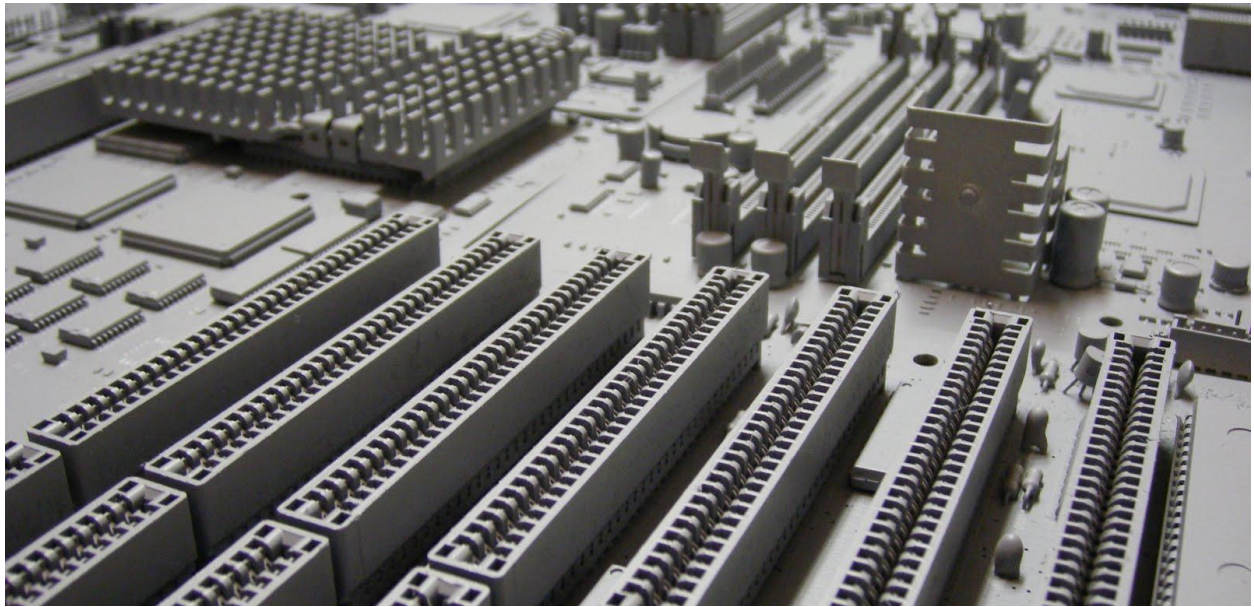
```
PS> $car = @{ color = "Gray"; model = "Corvette"; manufacturer =  
"Chevrolet"; numberOfDoors = 2 }
```

Accessing the desired value is done by writing the name of the table `$car` followed by dot (.) and the desired key. Or writing the name of the key inside quotes like if it were an index:

```
PS> $car.color
Gray
PS> $car['model']
Corvette
```

New keys and values can be added with the "Add" Method

```
PS> $car.Add('year', 2019)
PS> $car['year']
2019
```

Introduction

Operators are special **keywords and symbols** that work and transform data throughout the execution of the script.

In-Depth

Every single programming/scripting language has different keywords to represent pre-programmed actions for the programmer to use. Operators are mostly shared between languages to allow an easier transition from one to another, although this is not the case for Powershell, the reason for this is to maintain compatibility with all shells and environments.

Powershell has more operators than most languages, but it certainly helps when managing and transforming data.

Arithmetic Operators

These operators are quite simple to learn as we all know most of them already from **math**. There is one new and the ones that we know have new perks in Powershell allowing them to work with types that are not numerical nor text. Arithmetic operators:

- +
- -
- *
- /
- %

We will not list examples for the uses we already know, we think is fair to assume we all know how to add, subtract, multiply and divide numbers so let's see the Powershell uses for + and *, then we will explore % (Module):

- +
 - Concatenate Strings

```
PS>$fullPhrase = 'Hi ' + 'there'
PS>$fullPhrase
Hi there
```

- Concatenate Arrays

```
PS>$oneToFive = @(1,2,3) + @(4,5)
PS>$oneToFive
1
2
3
4
5
```

- Concatenate Hashtables

```
PS>$phoneCodes = @{'Argentina'='+54'} + @{'United
States'='+1'}
PS>$phoneCodes
```

Name	Value
----	-----
United States	+1
Argentina	+54

- *

- Add x times an element in an Array

```
PS>$arraysOfFourHellos = @('Hello!') * 4
PS>$arraysOfFourHellos
Hello!
Hello!
Hello!
Hello!
```

- Concatenate x times a String

```
PS>'aA' * 4
aAaAaAaA
```

Now that we've cleared all of the new uses for the known arithmetic operators we can focus on %, **the module operator**. This operator returns the remainder of a division:

```
PS>11 % 5
1
PS>11 % 3
2
```

This may not seem useful but we can see an easy example that is simplified by the use of %.

“Create a script that takes a value from the user and returns ‘PAIR’ if the value is pair and ‘ODD’ if the value is odd”:

Without %:

```
Param(
    [Parameter(Mandatory = $true, Position = 0)]
    [Int32]$number
)
$result = $number / 2
$pointPosition = $result.ToString().IndexOf('.')
if($pointPosition -ne -1){
    return 'ODD'
}else{
    return 'PAIR'
}
```

Execution:

```
PS> & '.\Operators - Arithmetic operators - Without module.ps1' 5
ODD
```

```
PS> & '.\Operators - Arithmetic operators - Without module.ps1' 4
PAIR
```

With %:

```
Param(
    [Parameter(Mandatory = $true, Position = 0)]
    [Int32]$number
)
if($number % 2 -ne 0){
    return "ODD"
}else{
    return "PAIR"
}
```

Execution:

```
PS> & '.\Operators - Arithmetic operators - With module.ps1' 7
ODD
```

```
PS> & '.\Operators - Arithmetic operators - With module.ps1' 10
PAIR
```

As we can see it's quite simpler to just use **%** to get the remainder instead of checking if the result is a Double or if it has a dot for the decimals.

Now we know all about the arithmetic operators.

Assignment Operators

As you might remember from Variables we have an infinite amount of types and values we can use in Powershell, given this, we have a comprehensive amount of assignment operators to work and **determine which value has to be assigned to a variable.**

Let's start with a math example to ease into this subject, having the following we can easily solve what is the value of 'X':

$$X = 5 - 3 \Rightarrow X = 2$$

This is clear for us because it's simple math, but for an interpreter¹ such as the Powershell console, there are many points to think before assigning 2. The first and most important is "Does X accept [Int32] values?" let us observe these different cases:

```
PS>$X = 5 - 3
PS>$X
2
```

Great no issues, but what if:

```
PS>[String]$X = 5 - 3
PS>$X
2
```

Awesome no errors again, until...:

```
PS>$Xplus5 = $X + 5
PS>$Xplus5
25
```

Wait that's not right, we were expecting 7, why is it 25? Is Powershell broken? Well no, it worked as you asked, it returned us 25 because you said X was supposed to

¹ *Interpreter: Powershell is not a compiled language so everything you run is executed by the interpreter, we could think of it like the brain of Powershell.*

contain a String and as we've previously seen the + operator concatenates Strings. Here we see how the = operator works. Let's explore it technically:

The = operator needs to determine if the value you want to assign is **acceptable** for the **type of variable** you are assigning it to. Our first example returned 2 as an Integer because of the arithmetic operator - does a mathematical subtraction of the numbers and returns a number, in this case, it's an Integer. But in our second example, we told the operator, \$X is a variable of type [String] so when it came time to store the Integer 2 it checked "Is this Type castable to a String?" and yes it is because we have 2 and "2" both the same for us because we know the difference out of the context we are using it. Let's check a couple of examples where the value is not castable:

```
PS>[bool]$X = 5 - 3
PS>$x
True
```

Here we see it took the 2 and assigned the boolean variable \$X to \$true because **any value not \$null is a \$true in boolean**, all and all still sort of castable, but here we see what happens in a more limited Type:

```
PS>[Hashtable]$X = 5 - 3
Cannot convert the "2" value of type "System.Int32" to type
"System.Collections.Hashtable".
At line:1 char:1
+ [Hashtable]$X = 5 - 3
+ ~~~~~
    + CategoryInfo          : MetadataError: (:) [],
ArgumentTransformationMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

Ha, we get an Exception saying *"Cannot convert the "2" value of type "System.Int32" to type "System.Collections.Hashtable"* and we can observe what we already knew, the value 2 returned by the subtraction is not castable to a Hashtable.

So the rule for an assignment operator can be simplified to:

<assignable-expression> <assignment-operator> <value>

Where the assignable-expression can contain or cast the received value.

Now that we have a technical definition of what an assignment operator is, we can continue with the other types of assignment operators. Some of these have their own inherent differences with certain types, We'll include an example for each difference.

These are all assignment operators:

- =
- +=
- -=
- *=
- /=
- %=
- ++
- --

We can divide these into 3 categories: **assignment**, **auto-arithmetic assignment** and **increment and decrement**. We've already explored the normal assignment operator now let's check the auto-arithmetic assignment with an example for each one:

- +=

```
PS>$number = 5
PS>$number += 10
PS>$number
15
```

- -=

```
PS>$number = 11
PS>$number -= 6
PS>$number
5
```

- *=

```
PS>$number = 6
PS>$number *= 3
PS>$number
18
```

- /=

```
PS>$number = 18
PS>$number /= 3
PS>$number
6
```

- %=

```
PS>$number = 11
PS>$number %= 5
PS>$number
1
```

As we can see from the examples it seems the auto-arithmetic assignments utilize their same value in the operation we also can write their equivalences:

`$number += 5` \Rightarrow `$number = $number + 5`

`$number -= 5` \Rightarrow `$number = $number - 5`

<assignable-expression> <auto-arithmetic assignment> <value>

|
∨

<assignable-expression> <assignment-operator> <assignable-expression>
<arithmetic operator> <value>

In simple words, it performs the arithmetic operation using the variable of assignment as the left value, therefore it never changes its type. It also applies all of the arithmetic functionalities to this assignment (I.e. a += can concatenate String, Arrays, and Hashtables)

Increment/Decrement Operator

And at last, the last type of assignment operators, the increment and decrement (++ and --). These are used for a very simple purpose but they are very widely used and quite frankly completely necessary. A quick example before the explanation:

```
PS>$number = 5
PS>$number++
PS>$number
6
```

```
PS>$number = 5
PS>$number--
PS>$number
4
```

As we can see it only added and subtracted 1, why would this be useful? It's just an abbreviation of $\$variable = \$variable \pm 1$. And yes it's just that until we see the different ways to apply this operator:

Pre-Increment / decrement:

```
PS>$a = 5
PS>$b = ++$a
PS>$b
6
PS>$a
6
```

```
PS>$a = 5
PS>$b = --$a
PS>$b
4
PS>$a
4
```

Post-Increment / decrement:

```
PS>$a = 5
PS>$b = $a++
PS>$b
5
PS>$a
6
```

```
PS>$a = 5
PS>$b = $a--
PS>$b
5
PS>$a
4
```

Now we see that this operator is position dependent, we can obtain different results on $\$b$ depending on where do we place the $++$ or $--$ in regards to $\$a$.

The difference between Pre and Post increment/decrement is **when the actual operation occurs**. In both examples the increment/decrement operators we used as part of an assignment operation the position determines if the increment/decrement should occur before performing the current operation or after

it has occurred, this is very useful because it gives you the **control and flexibility** to better manage your counter.

Comparison Operators

Here we find the biggest difference Powershell has compared to other programming/scripting languages, instead of adopting the usual comparison operators, shared by the most popular languages, it opted for creating completely new ones. Some of these operators mimic the functionality of traditional comparison operators and some are completely new. Here is a list of the basic comparison operators in Powershell:

- `-eq` Equals $\Rightarrow =$
- `-ne` Not equals $\Rightarrow \neq$
- `-gt` Greater than $\Rightarrow >$
- `-ge` Greater than or equal $\Rightarrow \geq$
- `-lt` Less than $\Rightarrow <$
- `-le` Less than or equal $\Rightarrow \leq$
- `-and` And $\Rightarrow \&\&$
- `-or` Or $\Rightarrow ||$
- `-not` Not $\Rightarrow !$

These comparison operators are always part of a **proposition** and they return its value of truth either `$true` or `$false`, for example:

```
PS>5 -eq 6
False
```

```
PS>5 -ne 6
True
```

```
PS>6 -gt 6
False
```

```
PS>6 -ge 6
True
```

```
PS>5 -lt 6
True
```

```
PS>7 -le 6
False
```

And just like all the other operators, they need to work with an infinite amount of types and values. So how do they behave when they are not used with numbers?

```
PS>$result = @(1,2,3) -eq @(4,5,6)
PS>$result
PS>
```

```
PS>$result = @{'Argentina'='+54'} -eq @{'United States'='+1'}
PS>$result
False
```

```
PS>$result = "hello" -eq "Hello"
PS>$result
True
```

So, we have 3 different results, all showing us something new about the comparison operators.

We see that the check for equality on Arrays returns \$null and it kind of makes sense, right? What were we checking? By size? Content? Content in a specified

position? How would the operator know? How do the operators know what to return? Because as we can see the Hashtable comparison returned False.

The way that the operators know how to compare each Type is defined in the Type itself in the class, but as this is quite complex, for now, let's just leave it at some objects know how to do some comparison and some don't. You'd just have to check beforehand, otherwise you can find yourself landing in an Exception:

```
PS>$result = @{'Argentina'='+54'} -ge @{'United States'='+1'}
Cannot compare "System.Collections.Hashtable" because it is not
IComparable.
At line:1 char:1
+ $result = @{'Argentina'='+54'} -ge @{'United States'='+1'}
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [],
RuntimeException
+ FullyQualifiedErrorId : NotIComparable
```

We will use and explain further this type of operators in the next subject, Flow Control.

There are many more comparison operators some designed to work with this more constrained Types (We will not explore these but FYI):

- -like *Returns true when string matches wildcard pattern*
- -notlike *Returns true when string does not match wildcard pattern*
- -match *Returns true when string matches regex pattern; \$matches contains matching strings*
- -notmatch *Returns true when string does not match regex pattern; \$matches contains matching strings*
- -contains *Returns true when reference value contained in a collection*
- -notcontains *Returns true when reference value not contained in a collection*
- -in *Returns true when test value contained in a collection*
- -notin *Returns true when test value not contained in a collection*
- -replace *Replaces a string pattern*
- -is *Returns true if both objects are the same type*
- -isnot *Returns true if the objects are not the same type*

Flow Control



Introduction

Flow Control are the keywords that allow the programmer to **choose** the desired **sentences to execute** depending on the truth value of the **propositions**.

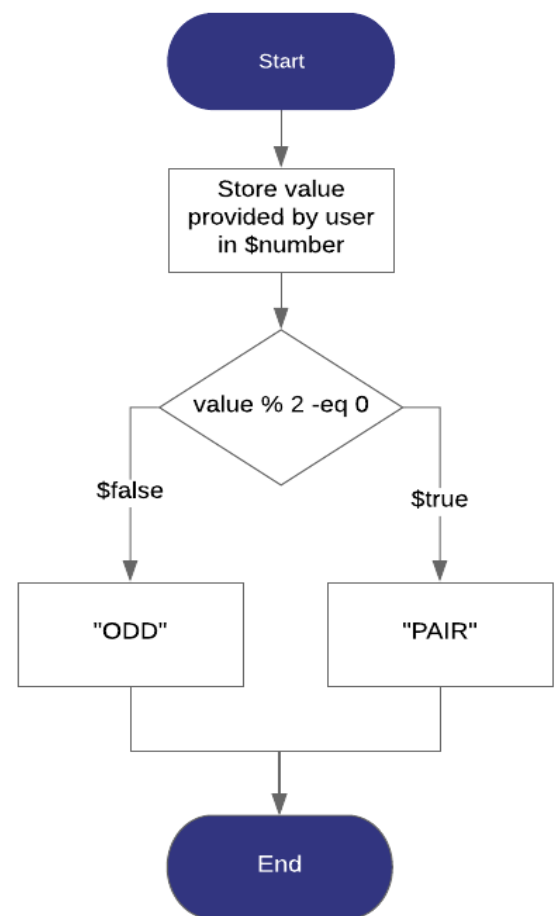
In-Depth

A simple script execution can be read from top to bottom on the script file, but for obvious reasons we need to select if and when to run parts of it, depending on the **state**². If we think this as an isolated module, for example, we can re-visit our previous example:

```
Param(  
    [Parameter(Mandatory = $true,  
    Position = 0)]  
    [Int32]$number  
)  
if($number % 2 -ne 0){  
    return "ODD"  
}else{  
    return "PAIR"  
}
```

Here we can see 2 desired outcomes for this script depending on the value of “\$number % 2 -ne 0”. Which returns \$true if pair and \$false if not, therefore choosing out outcome correctly.

We could express the **Flow** of this script the way it's expressed in the image, as we can see the flow is divided for part of the script and this is mandatory to fulfill the requirements.



² **State:** The collection of the variable's values at a chosen moment during the execution. I.e "Right after storing 2 into \$number we can say at the least, the current state is \$number=2"

If / If-Else / If-Elseif-Else

We've already seen the **if** conditional in action. In fact, we've seen the if-else in action too. This controller is designed in a way you could read its outcome in plain english. For instance in our previous example we could translate it to *"If the given number is divisible by 2 and the remainder is 0, we should write 'PAIR' to the screen otherwise we should write 'ODD' "*.

This controller's technical definition and syntax is:

```
if ([boolean]) {  
    #Run this if [boolean]$true  
} else {  
    #run this if [boolean]$false  
}
```

Remembering our previous chapter every comparison operator returns a boolean as a result, this determines how the flow of execution continues.

Now as almost everything conventional in programming Powershell adds new functionalities I.e:

```
PS>$number = 5  
>>  
>> if ($number) {  
>>     'The proposition is not $null'  
>> } else {  
>>     'The proposition is $null'  
>> }  
>>  
The proposition is not $null  
  
PS>if ($number) {  
>>     'The proposition is not $null'  
>> } else {  
>>     'The proposition is $null'  
>> }  
>>  
The proposition is $null  
PS>
```

Here we can see a **cast** to boolean, as you might remember from the Variables chapter, anything not \$null when casted, a boolean is \$true.

Awesome, so what do we do if we have more than 2 outcomes...? We could put an if inside the if, it's not the most elegant solution but it does the trick even though is frowned upon for a good reason (in other ways please avoid these type of weird solutions at all cost). We can think of a better way, introducing the IF-ELSE-IF-ELSE with this example

"Given 3 sides of a triangle output 'Equilateral', 'Scalene' and 'Isosceles' accordingly":

```
Param(
    [Parameter(Mandatory = $true, Position = 0)]
    [Int32] $side,
    [Parameter(Mandatory = $true, Position = 1)]
    [Int32] $side1,
    [Parameter(Mandatory = $true, Position = 2)]
    [Int32] $side2
)
if($side -eq $side1 -and $side1 -eq $side2){
    'Equilateral'
}elseif($side -eq $side1 -or $side1 -eq $side2 -or $side -eq
$side2){
    'Isosceles'
}else{
    'Scalene'
}
```

Execution:

```
PS> & '.\Flow Control - IF - Triangles.ps1' 1 2 3
Scalene
```

```
PS> & '.\Flow Control - IF - Triangles.ps1' 1 2 2
Isosceles
```

```
PS> & '.\Flow Control - IF - Triangles.ps1' 2 2 2
Equilateral
```

Now what size is the limit to keep adding elseif? well there is not an actual limit, BUT I believe 2 elseif is an acceptable level to maintain a readable script, in case more are necessary we should use a Switch which is the next subject to explore.

Switch

Usually on programming languages the Switch controller is a way to **compare a value against several propositions** and execute all that match or only the first. This is because a switch contains a flow of its own, if you execute a 'case' and you don't tell it to stop only with that it'll cascade to the bottom, this is the usual syntax and Powershell's:

C++:

```
switch ( test ) {
    case 1 :
        // Process for test = 1
        ...
        break;
    case 5 :
        // Process for test = 5
        ...
        break;
    default :
        // Process for all other cases.
        ...
}
```

Powershell:

```
switch (<value>){
    "String"    { statementlist }
    Number      { statementlist }
    Variable    { statementlist }
    {expression} { statementlist }
    default     { statementlist }
}
```

Right out the gate we notice the differences between the 2 implementations, even though both are based under the same idea they behave differently. For example, if you observe

the C++ implementation you'll notice the keyword 'break', this is used to move the flow out of the switch. Without this keyword all of the cases from the first that match to the end would be executed. Powershell's implementation doesn't have this problem as only 1 case would be executed, the common uses for a Switch are better explained with an example.

"A call center makes the users choose which menu they want by pressing a number between 0 and 4, create a switch to redirect them to the correct department. (Make up the department's names)":

```
Param(
    [Parameter(Mandatory = $true, Position = 0)]
    [Int32]$userInput
)
Switch ($userInput){
    0{
        'Talk to an operator'
    }
    1{
        'Sales - New Customers'
    }
    2{
        'Sales - Current Customers'
    }
    3{
        'Pending Inquiries'
    }
    4{
        'Complains'
    }
    default{
        "You've selected an invalid option"
    }
}
```

Execution:

```
PS> & '.\Flow Control - Switch - Call Center.ps1' 0
Talk to an operator
```

```
PS> & '.\Flow Control - Switch - Call Center.ps1' 3
Pending Inquiries
```

```
PS> & '.\Flow Control - Switch - Call Center.ps1' 7
You've selected an invalid option
```

Here we have a Switch in action. Notice the **default** option, a switch **should always have a default option**, even if you can't imagine a way it wouldn't match with the previous options. One of the holy commands of programming is that you can never trust a user's input, no matter how much you constraint it.

While

With the While we enter into a new category, the loops. A loop in programming as its name may denote, it's a controller that **maintains the flow in a set of sentences** over and over again until a condition is met, such as when you are making a coffee in the coffee machine and after you press the button you continually wait until the coffee is done and then you continue with your day.

The **syntax** for a while in Powershell is:

```
while (<condition>){  
    <statement list>  
}
```

Example:

```
PS>$number = 0  
PS>while($number -le 10){  
>>     $number += 2  
>>     $number  
>> }  
2  
4  
6  
8  
10  
12
```

As we can see the while controller can also be read as a phrase, *"While \$number is less than or equal to 10 add 2 to \$number and show it"*, that's how the previous example would be read. Now a more complicated example.

“Given an empty array fill it up starting from 1 to 200”:

```
$array = @()
$counter = 1
while($counter -ne 201){
    $array += $counter
    $counter ++
}
$array.length
```

Execution:

```
PS >& ‘.\Flow Control - While - Array to 200.ps1’
200
```

There you go just a couple of lines and you have the solution, you could also have done this by writing 200 statements or by initializing the array with 200 values, but that just wastes too much time, you should also use a loop **when recurring to the same sentence more than once**.

But as good as the while is, it has its shortcomings. You almost always have to declare a counter unless you are just waiting for a job to finish running, knowing this another controller was invented, the For loop our next topic.

For

The For loop can be described as a While with an embedded counter. This is the For controller’s syntax:

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

Init: First statement to be executed, generally the declaration of the counter variable (\$i = 0).

Condition: Proposition to evaluate truth value (\$i -le 10).

Repeat: Statement that runs at the end of every run, generally the increase of the counter variable (\$i++).

```
PS>for($i = 0;$i -le 10;$i++){
>>     $i
>> }
0
1
2
3
4
5
6
7
8
9
10
```

The While and the For loops are used also to go through arrays. As we know, you can choose a value from a position of an array by using the operator [], but if you need to work with more than one value in it, and most likely you do, you'll need to go value to value and you will need a counter to do it and a loop to run this code several times like this:

```
PS>$array = @("Hello ", "my ", "friend.")
PS>$completeString = ''
PS>for($i = 0;$i -lt $array.length;$i++){
>>     $completeString += $array[$i]
>> }
PS>$completeString
Hello my friend.
```

With the While and For loops you can choose whichever you want, there isn't really a general example to choose For instead of While or vice versa, unless you want to work with objects, and in Powershell you always can and do, by using the ForEach.

ForEach

The ForEach is a controller created to **go through Collections**, no counter necessary, this is completely necessary in Powershell as everything is an object. This is the ForEach syntax:

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

The way it works is taking one item from the collection at a time and using it in the statement list, we can explore this with the following example:

“Given a set of processes find the one with the highest CPU usage”:

```
$processes = Get-Process  
$max = $processes[0]  
foreach($process in $processes){  
    if($process.CPU -gt $max.CPU){  
        $max = $process  
    }  
}  
$max
```

Execute:

```
PS> & '.\Flow Control - ForEach - Sort by CPU usage.ps1'
```

Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	--	--	-----
554	28	15116	57680	3,095.42	11100	1	LockApp

As we can see the foreach went through the processes one by one, compared it with the last max without the need for counters and the operator []. This collector works with every type of collection in the .Net framework.

Cmdlets & the Pipeline

Get-WindowsImageContent	Cmdlet	Dism	Get-WindowsImageContent...
Get-DnsClient	Function	DnsClient	...
Get-DnsClientCache	Function	DnsClient	...
Get-DnsClientServerAddress	Function	DnsClient	...
Get-DnsClientNrptGlobal	Function	DnsClient	...
Get-DnsClientNrptPolicy	Function	DnsClient	...
Get-DnsClientNrptRule	Function	DnsClient	...
Get-DnsClientGlobalSetting	Function	DnsClient	...
Get-WinLanguageBarOption	Cmdlet	International	Get-WinLanguageBarOption...
Get-WinCultureFromLanguageList...	Cmdlet	International	Get-WinCultureFromLanguageListOptOut...
Get-WinHomeLocation	Cmdlet	International	Get-WinHomeLocation...
Get-WinSystemLocale	Cmdlet	International	Get-WinSystemLocale...
Get-WinAcceptLanguageFromLangu...	Cmdlet	International	Get-WinAcceptLanguageFromLanguageListOptOut...
Get-WinUILanguageOverride	Cmdlet	International	Get-WinUILanguageOverride...
Get-WinUserLanguageList	Cmdlet	International	Get-WinUserLanguageList...
Get-WinDefaultInputMethodOverride	Cmdlet	International	Get-WinDefaultInputMethodOverride...
Get-IscsiConnection	Function	iSCSI	...
Get-IscsiSession	Function	iSCSI	...
Get-IscsiTarget	Function	iSCSI	...
Get-IscsiTargetPortal	Function	iSCSI	...
Get-IseSnippet	Function	ISE	...
Get-KdsConfiguration	Cmdlet	Kds	Get-KdsConfiguration...
Get-KdsRootKey	Cmdlet	Kds	Get-KdsRootKey...
Get-WinEvent	Cmdlet	Microsoft.PowerShell.D...	Get-WinEvent...
Get-Counter	Cmdlet	Microsoft.PowerShell.D...	Get-Counter...
Get-LocalGroupMember	Cmdlet	Microsoft.PowerShell.L...	Get-LocalGroupMember...
Get-LocalGroup	Cmdlet	Microsoft.PowerShell.L...	Get-LocalGroup...
Get-LocalUser	Cmdlet	Microsoft.PowerShell.L...	Get-LocalUser...
Get-Credential	Cmdlet	Microsoft.PowerShell.S...	Get-Credential...
Get-Acl	Cmdlet	Microsoft.PowerShell.S...	Get-Acl...
Get-CmsMessage	Cmdlet	Microsoft.PowerShell.S...	Get-CmsMessage...
Get-PfxCertificate	Cmdlet	Microsoft.PowerShell.S...	Get-PfxCertificate...
Get-AuthenticodeSignature	Cmdlet	Microsoft.PowerShell.S...	Get-AuthenticodeSignature...
Get-ExecutionPolicy	Cmdlet	Microsoft.PowerShell.S...	Get-ExecutionPolicy...
Get-WSManInstance	Cmdlet	Microsoft.WSMan.Manage...	Get-WSManInstance...
Get-WSManCredSSP	Cmdlet	Microsoft.WSMan.Manage...	Get-WSManCredSSP...

Introduction

These topics are the **most important features** of Powershell, both of them make the Powershell environment so flexible and compatible with almost every solution you want to implement.

Cmdlets are **snippets of code** that englobe a **single functionality**. For example let's take one that we've already used, Get-Process. As its name suggests, it's designed to get the processes that are currently running on the computer, when you run it an underlying piece of code hidden to you is completing the task and returning you the results.

The Pipeline is a tool that allows you to **connect the results** from one Cmdlet or Operation to another. As the name suggests, think this as a pipe where the resultant object continues to the next exit point.

In-Depth

The technological definitions of these two features are far too complicated to explain at this level.

Cmdlets

There are several hundreds of Cmdlets pre installed in Powershell out of the box, and you can add more with Modules (explained in the next class), so it's impossible to know them all by heart. There needs to be a system in place for you to find the Cmdlet you want fast and easy, well there is so let's explore it.

Powershell Cmdlets are composed of a Verb + Substantive (I.e, Get - Process). That way you can think of what you want to do, for example, "I want to get all of the variables in this session", how do you think that command is named? . . . if you thought "Get-Variable" you were right.

Now, how do you know if you are correct and that Cmdlet will do exactly what you wanted it to do? This brings us to our first primordial Cmdlet, Get-Help.

Get-Help

As the name suggests, it is used to get help about something. But how do you say to the Cmdlet *"I want help about this X topic"*? Well, Cmdlets have things called **Parameters** that compose the **Intake of information** for it to perform its task.

You can find out the parameters you have to work with by following the next steps:

1. Write the Cmdlet you want to explore

```
PS>Get-Help
```

2. Add a space after the Cmdlet or last Parameter and write a '-' character

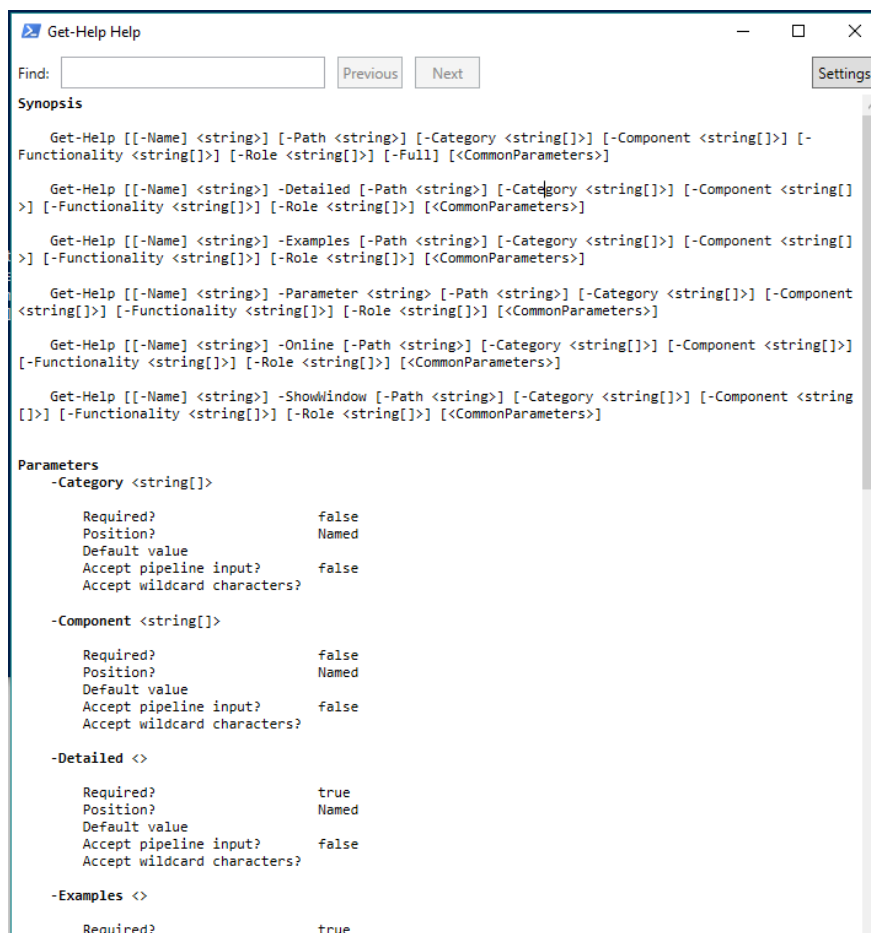
```
PS>Get-Help -
```

3. Press the Tab key repeatedly until you find the parameter you want

```
PS>Get-Help -Name
PS>Get-Help -Path
PS>Get-Help -Category
```

Great, we've found the parameter, or at least some of them. Some of these parameters are shared between different Cmdlets to create constant functionality through all Cmdlets in Powershell. This are the easiest to learn how to use/what they do, but how do we find out what the other parameters do? Well, you use Get-Help, Specifically:

```
PS>Get-Help -Name 'Get-Help' -Full
or
PS>Get-Help -Name 'Get-Help' -ShowWindow
```



With both you can see the entire description of all the parameters and all the ways to complete and use the cmdlet also it gives you a link that you can follow to the Powershell documentation for some examples in the use of the cmdlet.

Now that we've seen how to manage a Cmdlet let's explore the rest of what I consider primordial.

Set-Location

This Cmdlet and its counterparts in different languages is one of the most used cmdlets because it allows you to explore the filing system of the computer. For example, let's say you want to see all the files that are inside a folder, you could write the entire path to it or enter the folder and just run the Cmdlet without parameters:

```
PS>Set-Location W:\Code\Scripts\Powershell\  
PS>Get-Location
```

```
Path  
----  
W:\Code\Scripts\Powershell
```

```
PS>Get-ChildItem
```

```
Directory: W:\Code\Scripts\Powershell
```

Mode	LastWriteTime		Length	Name
----	-----		-----	----
d-----	9/8/2018	6:24 PM		.vs
d-----	10/24/2018	8:39 PM		ICOs
d-----	10/24/2018	8:39 PM		Learning
d-----	9/8/2018	6:24 PM		Modules
d-----	3/21/2019	12:55 PM		Projects
d-----	6/21/2019	11:36 PM		ps1
d-----	10/25/2018	10:00 PM		Test
d-----	10/25/2018	10:26 PM		To be reviewed
d-----	3/21/2019	12:55 PM		Tools
-a-----	10/25/2018	10:29 PM	200	.gitignore
-a-----	10/25/2018	7:46 PM	72	README.md

Alias for Set-Location: 'sl' or 'cd'

Get-ChildItem

We used this Cmdlet in our previous example and it's pretty much self explanatory. What it does is, it returns you the child (files) under the parent (Folder), with all of their properties.

```
PS>Get-ChildItem
```

```
Directory: W:\Code\Scripts\Powershell
```

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d-----	9/8/2018	6:24 PM		.vs
d-----	10/24/2018	8:39 PM		ICOs
d-----	10/24/2018	8:39 PM		Learning
d-----	9/8/2018	6:24 PM		Modules
d-----	3/21/2019	12:55 PM		Projects
d-----	6/21/2019	11:36 PM		ps1
d-----	10/25/2018	10:00 PM		Test
d-----	10/25/2018	10:26 PM		To be reviewed
d-----	3/21/2019	12:55 PM		Tools
-a----	10/25/2018	10:29 PM	200	.gitignore
-a----	10/25/2018	7:46 PM	72	README.md

Alias for Get-ChildItem: 'ls' or 'dir'

Out-GridView

We've seen this Cmdlet in action on the "What is Powershell?" section, this Cmdlet creates a user interface easily navigable and manageable, let's use our previous example:

```
PS>Get-ChildItem | Out-GridView
```

Get-ChildItem | Out-GridView

Filter

+ Add criteria

Mode	LastWriteTime	Length	Name
d----	9/8/2018 6:24:24 PM		.vs
d----	10/24/2018 8:39:19 PM		ICOs
d----	10/24/2018 8:39:29 PM		Learning
d----	9/8/2018 6:24:25 PM		Modules
d----	2/24/2019 7:17:49 PM		New script gts
d----	3/21/2019 12:55:44 PM		Projects
d----	6/21/2019 11:36:16 PM		ps1
d----	10/25/2018 10:26:46 PM		PwC
d----	10/25/2018 10:00:50 PM		Test
d----	10/25/2018 10:26:46 PM		To be reviewed
d----	3/21/2019 12:55:44 PM		Tools
-a----	10/25/2018 10:29:13 PM	200	.gitignore
-a----	10/25/2018 7:46:24 PM	72	README.md

You can filter and re order this information, also you can copy and paste it into an excel or txt.

Read-Host

You use this Cmdlet in your scripts to ask for user input in text form, for example:

```
PS>$name = Read-Host -Prompt "What is your name?"
What is your name?: Computer
PS>$name
Computer
```

You can also use it to ask for passwords (even though there are much better ways..) adding the parameter '-AsSecureString' and you get an encrypted password:

```
PS>$password = Read-Host -Prompt "What is your password?"
-AsSecureString
What is your password?: *****
PS>$password | ConvertFrom-SecureString
01000000d08c9ddf0115d1118c7a00c04fc297eb010000000e525af451eeb94b8b93b816
5716105d0000000002000000000010660000000100002000000044d4027263a9bcd865bd
3e7a61150c687a6fb1cfeeff57a073a04d24ef8836ac000000000e800000000200002000
00000c8298890e571c2277dce6e6f5d6213c21b5e31d56588d23e8581dab7e36784b3000
00000e0cfdae89b7533b7a0325faa85f4e7ad8845775814d98116f4067eb28a8bc5d6d76
a269ee182ede7399ed55514fe44b400000004b17bc2dccbad160a2bb5e3a3c914f20a1bd
eccf1019eac7bba26118e25fbc0f0d09ea4bdc59590d13b1258cd316f952fd054eec0d9b
73ec2222ce36dcf798d5
```

Get-Member / .getType()

As we've already established several times, everything in Powershell is an object and every object is derived from a class that, as a blueprint for a house, determines what properties it has. To know what properties an object has in Powershell we use Get-Member. To know what the exact Type/Class the object is derived from we use .getType() (This not a Cmdlet it is a method, the action of an object. This will be explained further in the next class). Let's explore further our example with the files:

```
PS>Get-ChildItem | Get-Member
```

```
TypeName: System.IO.DirectoryInfo
```

Name	MemberType	Definition
----	-----	-----
LinkType	CodeProperty	System.String
LinkType{get=GetLinkType;}		
.	.	.
.	.	.
.	.	.

```
TypeName: System.IO.FileInfo
```

Name	MemberType	Definition
----	-----	-----
LinkType	CodeProperty	System.String
LinkType{get=GetLinkType;}		
.	.	.
.	.	.
.	.	.

As Get-ChildItem returns more than one Type of object Get-Member lists all of the types and its members, we can use the '.' operator to access all of these members, for example:

```
PS>$childItem = (Get-ChildItem)[12]
PS>$childItem.ToString()
README.md
PS>$childItem.Directory
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d-----	1/30/2019 12:53 PM		Powershell

```
PS>$childItem.FullName
W:\Code\Scripts\Powershell\README.md
PS>$childItem.Exists
True
PS>
```

And we can find the Type by using `GetType()` (*All objects on the .Net framework have this method*):

```
PS>$childItem.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     FileInfo
System.IO.FileSystemInfo
```

```
PS>
```

Alias for Get-Member: 'gm'

Pipeline and Data Filtering

You may have noticed the symbol "`|`" on the examples through this section. This is what in Powershell we called a pipeline. As previously explained in simple words, it is a way to **send the result from a Cmdlet to another**.

The Pipeline is the most powerful feature of Powershell and one of the hardest to master but at a beginner level is easy to use. We've seen several examples of commands that can be used without the pipeline now lets see some that are almost completely used in conjunction with it.

Select-Object

This Cmdlet is used to select **properties** from the object that passes through the pipeline, you can also select the first/last x amount of object that passed through the pipeline:

```
PS>Get-ChildItem | select Name
```

```
Name
----
.vs
ICOs
Learning
Modules
README.md
```

```
PS>Get-ChildItem | Select-Object -Last 3
```

```
Directory: W:\Code\Scripts\Powershell
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d-----	3/21/2019 12:55 PM		Tools
-a----	10/25/2018 10:29 PM	200	.gitignore
-a----	10/25/2018 7:46 PM	72	README.md

```
PS>Get-ChildItem | Select-Object -First 3
```

```
Directory: W:\Code\Scripts\Powershell
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d-----	9/8/2018 6:24 PM		.vs
d-----	10/24/2018 8:39 PM		ICOs
d-----	10/24/2018 8:39 PM		Learning

And the most used one is Select, to fully explore the object, and or, find the property I need:

```
PS>(Get-ChildItem)[0] | Select-Object *
```

```
PSPath           :
Microsoft.PowerShell.Core\FileSystem::W:\Code\Scripts\Powershell\.vs
PSParentPath      :
Microsoft.PowerShell.Core\FileSystem::W:\Code\Scripts\Powershell
PSChildName       : .vs
PSDrive           : W
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : True
Mode              : d-----
BaseName          : .vs
Target            : {}
LinkType          :
Name              : .vs
FullName          : W:\Code\Scripts\Powershell\.vs
Parent            : Powershell
Exists            : True
Root              : W:\
Extension         : .vs
CreationTime      : 9/7/2018 5:49:46 PM
CreationTimeUtc   : 9/7/2018 8:49:46 PM
LastAccessTime    : 9/8/2018 6:24:24 PM
LastAccessTimeUtc : 9/8/2018 9:24:24 PM
LastWriteTime     : 9/8/2018 6:24:24 PM
LastWriteTimeUtc  : 9/8/2018 9:24:24 PM
Attributes        : Directory
```

Alias of Select-Object: select

Where-Object

This Cmdlet is used to filter which information passes to the next level of the pipe by either filtering by a property or by proposing a proposition to test the truth value of.

For example,

"let's take our previous example and filter the files avoiding the folders and then show me their names only.":

```
PS>gci
```

```
Directory: W:\Code\Scripts\Powershell\ps1
```

Mode	LastWriteTime		Length	Name
----	-----		-----	----
d-----	10/25/2018	8:38 PM		.vs
d-----	9/8/2018	6:24 PM		Add-AllComputersToADGroup
d-----	9/27/2018	9:30 PM		Challenge goLang
d-----	9/8/2018	6:24 PM		Check ad groups
d-----	10/25/2018	10:06 PM		Check-AdGroupsOf
d-----	2/6/2019	11:15 AM		CRIS
d-----	9/8/2018	6:24 PM		ExploreFolders
d-----	9/8/2018	6:24 PM		FoldersArchive
d-----	9/8/2018	6:24 PM		GPOs
d-----	10/29/2018	7:47 PM		Notebook_Backup
d-----	7/14/2019	2:39 PM		PowerShell - Course
d-----	11/17/2018	11:56 AM		Project Euler
d-----	9/8/2018	6:24 PM		Sent
d-----	9/8/2018	6:24 PM		StyleWriter
d-----	11/17/2018	11:56 AM		Test Cristian
-a----	3/21/2019	12:55 PM	149	Get-IP.ps1
-a----	7/4/2017	7:10 AM	1045	Is-Expired.ps1
-a----	4/3/2018	12:45 PM	9609	RemoveAdmin.ps1
-a----	7/5/2017	5:15 AM	1869	Reset-Password.ps1
-a----	7/5/2017	5:24 AM	1869	Reset-Password_v2.ps1

```
PS>Get-ChildItem | Where-Object Attributes -eq "Archive" | select Name
-ExpandProperty Name
Get-IP.ps1
Is-Expired.ps1
RemoveAdmin.ps1
Reset-Password.ps1
Reset-Password_v2.ps1
Set-PasswordToRequireChange.ps1
WifiRoaming_wAdminrequest.ps1
And now one with a proposition.
```

“Select only the ones that start name’s start with an ‘R’”:

```
PS>Get-ChildItem | Where-Object {$_.Name.StartsWith("R")} | select Name
-ExpandProperty Name
RemoveAdmin.ps1
Reset-Password.ps1
Reset-Password_v2.ps1
```

** \$_: This is an auto generated variable that contains the object that just went through the pipeline, so we can use its properties.*

ForEach-Object

This Cmdlet takes an object from the pipeline and runs statements with it, just like the original **foreach**, the collection is divided by the pipeline and the ForEach-Object Cmdlet works with one at a time. Let's use it in an example:

"Create a folder for each String that passes through the pipeline":

```
PS>"Folder 1","Folder2","I created this folder with powershell" |  
ForEach-Object {New-Item -Name $_ -Type Directory}
```

```
Directory: W:\Code\Scripts\Powershell\ps1\PowerShell - Course
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d-----	7/14/2019 7:56 PM		Folder 1
d-----	7/14/2019 7:56 PM		Folder2
d-----	7/14/2019 7:56 PM		I created this folder with powershell

Outputs

With all of these tools you can very easily filter and manage large amounts of data. Now, what do you do when you have to filter this data? There are several outputs you can choose in Powershell, here is a list:

```
PS>Get-Command "ConvertTo-*" | select CommandType,Name,Version
```

CommandType	Name	Version
-----	----	-----
Function	ConvertTo-ExcelXlsx	5.4.5
Cmdlet	ConvertTo-Csv	3.1.0.0
Cmdlet	ConvertTo-Html	3.1.0.0
Cmdlet	ConvertTo-Json	3.1.0.0
Cmdlet	ConvertTo-ProcessMitigationPolicy	1.0.11
Cmdlet	ConvertTo-SecureString	3.0.0.0
Cmdlet	ConvertTo-TpmOwnerAuth	2.0.0.0
Cmdlet	ConvertTo-Xml	3.1.0.0

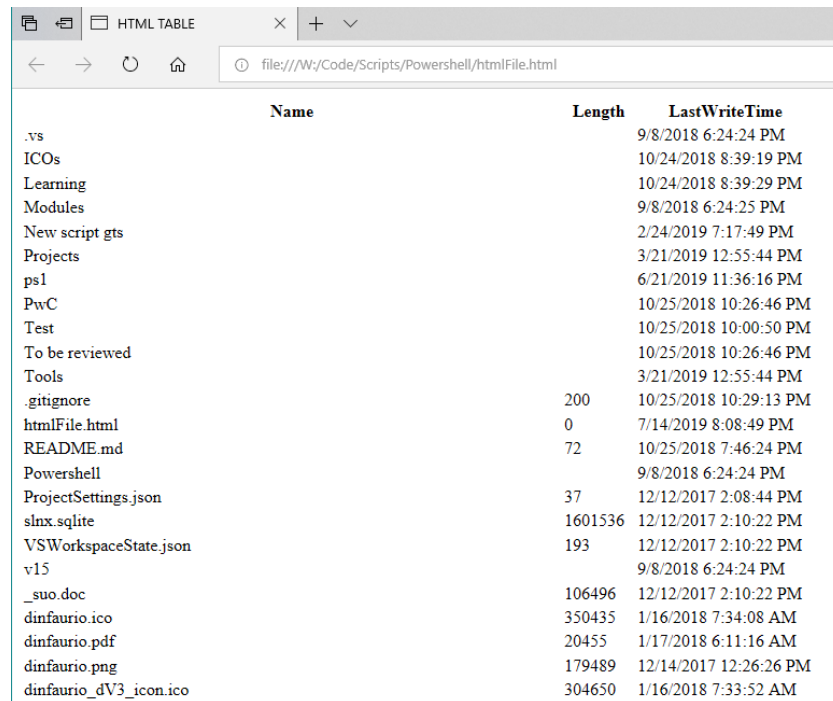
```
PS>Get-Command "Out-*" | select CommandType,Name,Version
```

CommandType	Name	Version
-----	----	-----
Cmdlet	Out-Default	3.0.0.0
Cmdlet	Out-File	3.1.0.0
Cmdlet	Out-GridView	3.1.0.0
Cmdlet	Out-Host	3.0.0.0
Cmdlet	Out-Null	3.0.0.0
Cmdlet	Out-Printer	3.1.0.0
Cmdlet	Out-String	3.1.0.0

And here is an Example:

“Show me a list of all the files in a folder in HTML format”:

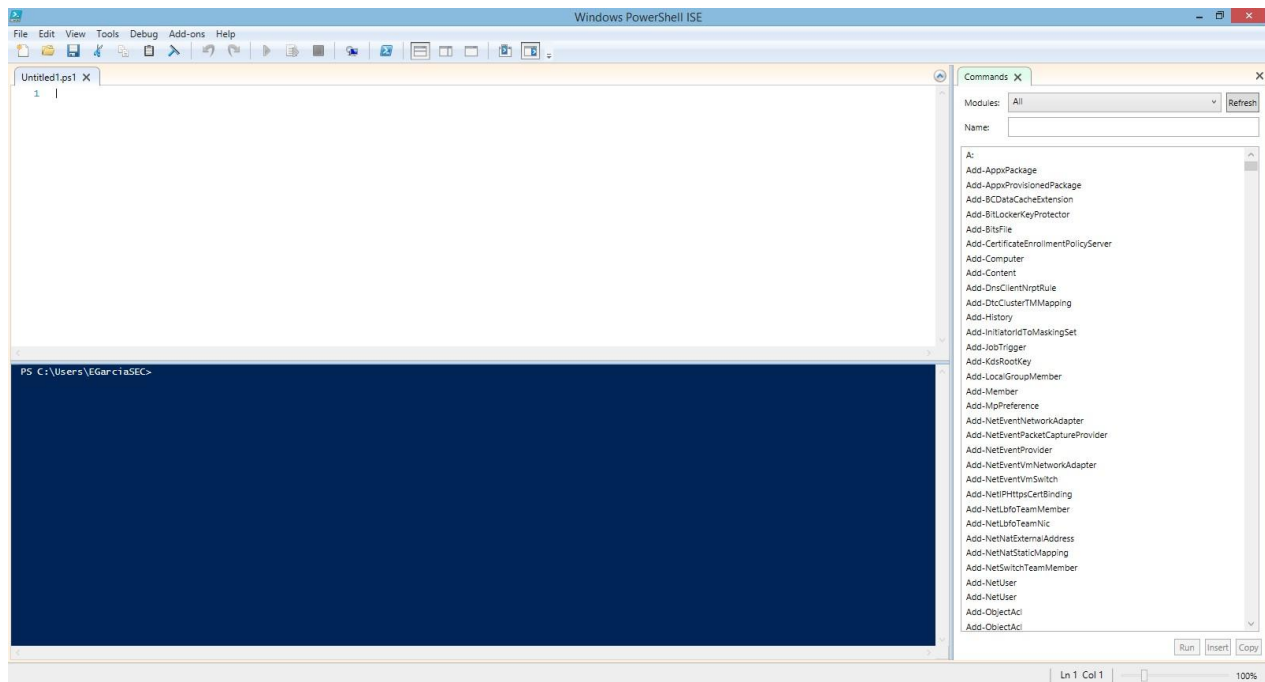
```
PS> gci -Recurse | select Name,Length,LastWriteTime | ConvertTo-Html |  
Out-File htmlFile.html
```



The screenshot shows a web browser window with the title 'HTML TABLE'. The address bar displays the file path: `file:///W:/Code/Scripts/Powershell/htmlFile.html`. The browser content is an HTML table with three columns: 'Name', 'Length', and 'LastWriteTime'. The table lists various files and folders in a directory, including .vs, ICOs, Learning, Modules, New script gts, Projects, ps1, PwC, Test, To be reviewed, Tools, .gitignore, htmlFile.html, README.md, Powershell, ProjectSettings.json, slnx.sqlite, VSWorkspaceState.json, v15, _suo.doc, dinfaurio.ico, dinfaurio.pdf, dinfaurio.png, and dinfaurio_dV3_icon.ico. The 'Length' column shows values for files, and the 'LastWriteTime' column shows the date and time of the last modification.

Name	Length	LastWriteTime
.vs		9/8/2018 6:24:24 PM
ICOs		10/24/2018 8:39:19 PM
Learning		10/24/2018 8:39:29 PM
Modules		9/8/2018 6:24:25 PM
New script gts		2/24/2019 7:17:49 PM
Projects		3/21/2019 12:55:44 PM
ps1		6/21/2019 11:36:16 PM
PwC		10/25/2018 10:26:46 PM
Test		10/25/2018 10:00:50 PM
To be reviewed		10/25/2018 10:26:46 PM
Tools		3/21/2019 12:55:44 PM
.gitignore	200	10/25/2018 10:29:13 PM
htmlFile.html	0	7/14/2019 8:08:49 PM
README.md	72	10/25/2018 7:46:24 PM
Powershell		9/8/2018 6:24:24 PM
ProjectSettings.json	37	12/12/2017 2:08:44 PM
slnx.sqlite	1601536	12/12/2017 2:10:22 PM
VSWorkspaceState.json	193	12/12/2017 2:10:22 PM
v15		9/8/2018 6:24:24 PM
_suo.doc	106496	12/12/2017 2:10:22 PM
dinfaurio.ico	350435	1/16/2018 7:34:08 AM
dinfaurio.pdf	20455	1/17/2018 6:11:16 AM
dinfaurio.png	179489	12/14/2017 12:26:26 PM
dinfaurio_dV3_icon.ico	304650	1/16/2018 7:33:52 AM

Integrated Scripting Environment (ISE)



Introduction

The Windows Powershell Integrated Scripting Environment (ISE) is a host application for Windows Powershell. In the ISE, you can run commands and write, test, and debug scripts in a single Windows-based graphic user interface.

The ISE provides multiline editing, tab completion, syntax coloring, selective execution, context-sensitive help and many other functionalities. Menu items and keyboard shortcuts are mapped to many of the same tasks that you would do in the Windows Powershell console. For example, when you debug a script in the ISE, you can right-click on a line of code in the edit pane to set a breakpoint.

In-Depth

Working with the ISE makes it a lot easier and faster to write down Scripts, as the syntax coloring and the sentence auto-completion allows you to code both faster and more consistently, eliminating the risk of typos almost entirely.

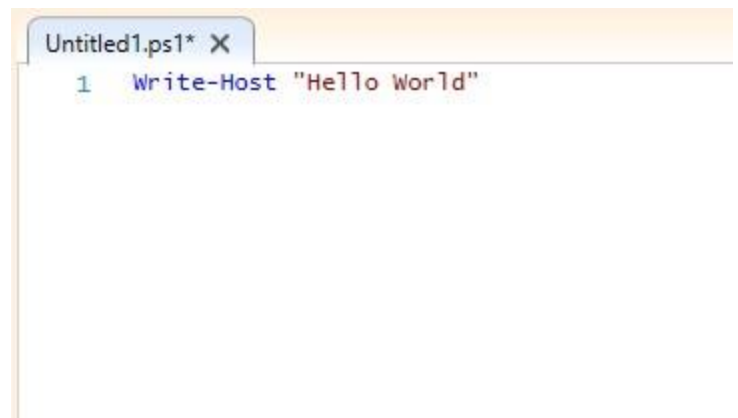
The image showed previously can be divided into three important sections

Script Editing

Here the script being written can be seen, and run. Running the Script can be done by either pressing F5 or the Run Button

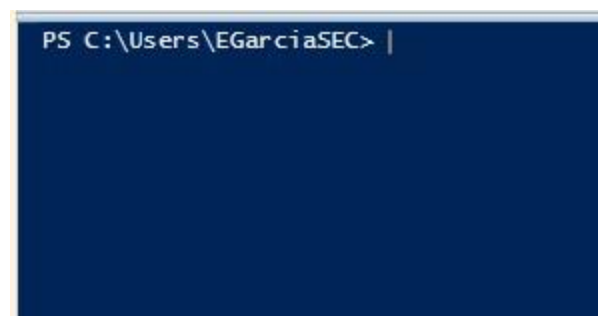


The ISE also allows the user to run the highlighted lines by pressing F8 or right clicking and choosing "Run Selection"



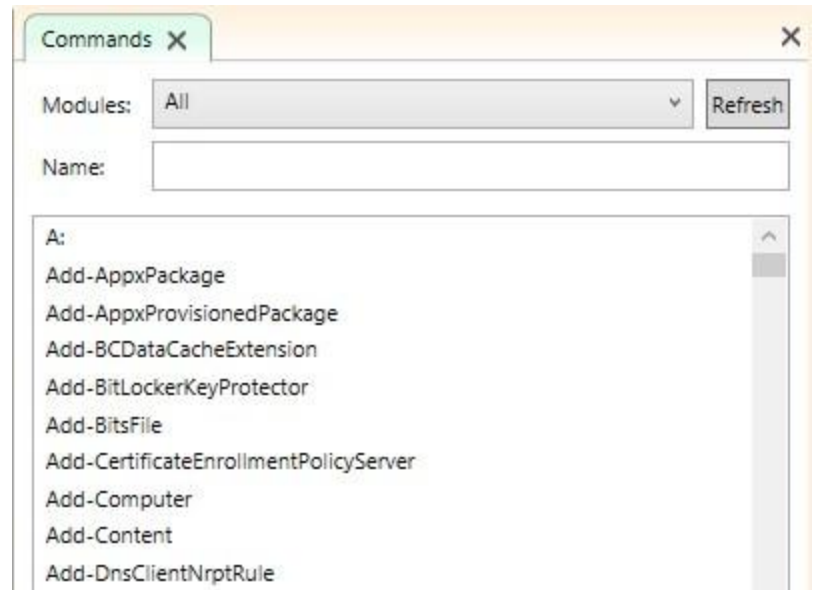
Integrated Console

Commands can be run directly in this console as with the usual Powershell Console. Do keep in mind that variables are kept in memory upon being initialized.



Command Add-On

With this useful tool you can look for and insert desired commands into your Script, it works like the Get-Help Command but usable through a User Interface



Debugging

The act of debugging, is probably the single most important step of getting to know what's going wrong with your Script. Debugging is the process of finding and resolving defects in your script by running it and analyzing the failing steps.

Debugging consists on the following steps:

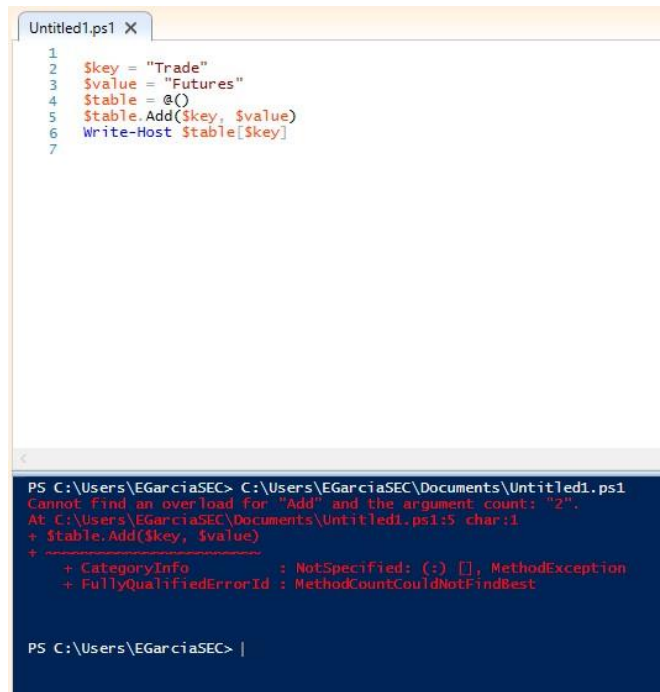
- Reproducing the Bug (the failing feature).
- Examine Program States (running the debugger tool to check the state in the suspected failing steps).
- Track down the lines of code causing the bug.
- Refactor code to prevent the undesired effects.

Debugging can be done on the ISE by setting breakpoints (steps where the script will temporarily stop).

These breakpoints can be set by pressing F9 or right clicking and selecting "Toggle Breakpoint". Next time you run the script it will stop at these designated steps. While the

script is stopped, other commands can be run That way you can check variable values, properties and run other useful commands.

See for example, we have this failing Script, which doesn't allow us to Add arguments to a Hash Table



The image shows a PowerShell script editor window titled 'Untitled1.ps1 X' with the following code:

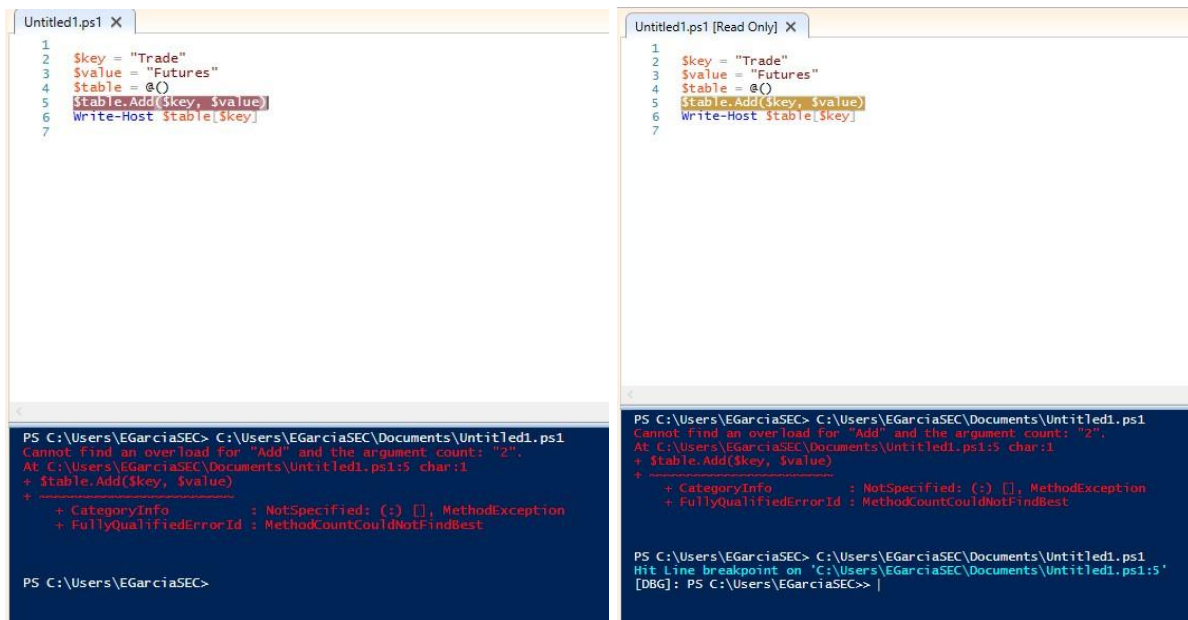
```
1 $key = "Trade"
2 $value = "Futures"
3 $table = @{}
4 $table.Add($key, $value)
5 Write-Host $table[$key]
```

Below the editor is a PowerShell console window showing the execution of the script and the resulting error:

```
PS C:\Users\EGarciaSEC> C:\Users\EGarciaSEC\Documents\Untitled1.ps1
Cannot find an overload for "Add" and the argument count: "2".
At C:\Users\EGarciaSEC\Documents\Untitled1.ps1:5 char:1
+ $table.Add($key, $value)
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodException
+ FullyQualifiedErrorId : MethodCountCouldNotFindBest

PS C:\Users\EGarciaSEC> |
```

We set a breakpoint on the line that's generating the issue and run the application again



The left screenshot shows a PowerShell script in a file named 'Untitled1.ps1'. The script contains the following code:

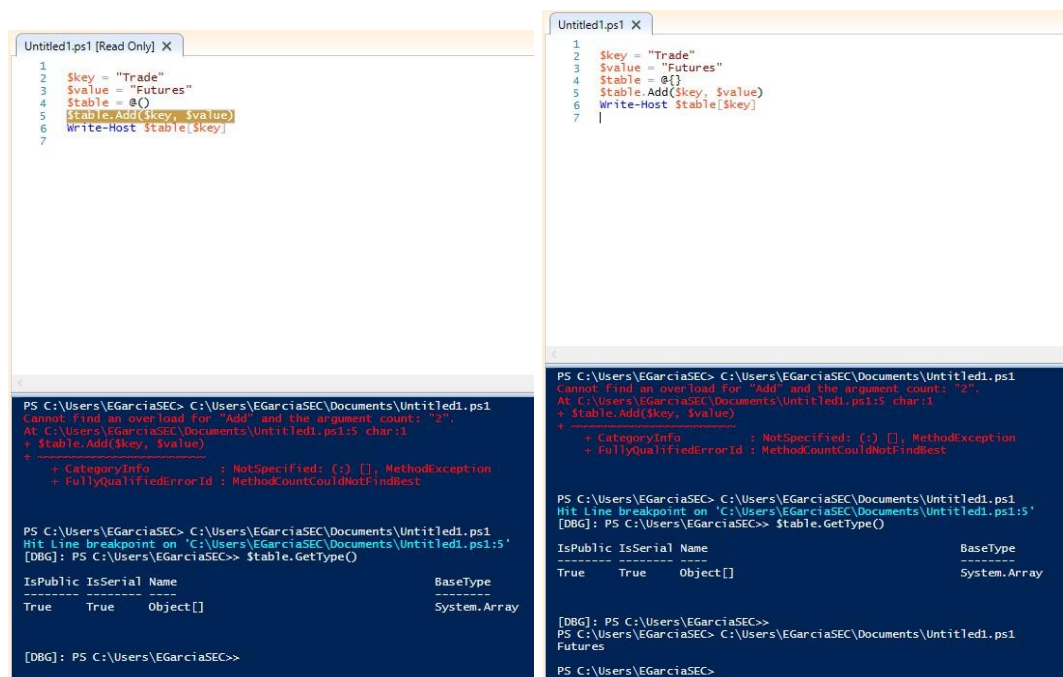
```
1 $key = "Trade"
2 $value = "Futures"
3 $table = @{}
4 $table.Add($key, $value)
5 Write-Host $table[$key]
```

The right screenshot shows the same script with a breakpoint set on line 4. The console output shows the error message:

```
PS C:\Users\EGarciaSEC> C:\Users\EGarciaSEC\Documents\Untitled1.ps1
Cannot find an overload for "Add" and the argument count: "2".
At C:\Users\EGarciaSEC\Documents\Untitled1.ps1:5 char:1
+ $table.Add($key, $value)
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodException
+ FullyQualifiedErrorId : MethodCountCouldNotFindBest

PS C:\Users\EGarciaSEC>
```

As you can see now we can run commands while the Script is stopped. Let's try checking the Type of \$table:



The left screenshot shows the same PowerShell script with a breakpoint set on line 4. The console output shows the error message:

```
PS C:\Users\EGarciaSEC> C:\Users\EGarciaSEC\Documents\Untitled1.ps1
Cannot find an overload for "Add" and the argument count: "2".
At C:\Users\EGarciaSEC\Documents\Untitled1.ps1:5 char:1
+ $table.Add($key, $value)
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodException
+ FullyQualifiedErrorId : MethodCountCouldNotFindBest

PS C:\Users\EGarciaSEC> C:\Users\EGarciaSEC\Documents\Untitled1.ps1
Hit Line breakpoint on 'C:\Users\EGarciaSEC\Documents\Untitled1.ps1:5'
[DBG]: PS C:\Users\EGarciaSEC> $table.GetType()

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

[DBG]: PS C:\Users\EGarciaSEC>
```

The right screenshot shows the same script with a breakpoint set on line 4. The console output shows the error message and the type of \$table:

```
PS C:\Users\EGarciaSEC> C:\Users\EGarciaSEC\Documents\Untitled1.ps1
Cannot find an overload for "Add" and the argument count: "2".
At C:\Users\EGarciaSEC\Documents\Untitled1.ps1:5 char:1
+ $table.Add($key, $value)
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodException
+ FullyQualifiedErrorId : MethodCountCouldNotFindBest

PS C:\Users\EGarciaSEC> C:\Users\EGarciaSEC\Documents\Untitled1.ps1
Hit Line breakpoint on 'C:\Users\EGarciaSEC\Documents\Untitled1.ps1:5'
[DBG]: PS C:\Users\EGarciaSEC> $table.GetType()

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

[DBG]: PS C:\Users\EGarciaSEC>
PS C:\Users\EGarciaSEC> C:\Users\EGarciaSEC\Documents\Untitled1.ps1
Futures

PS C:\Users\EGarciaSEC>
```

It's an array! Since it's not a Hash Table we can't add a key and value, so if we stop the process and change the value of \$table to @{} everything works as expected.

Conclusion

With this we conclude our first class on Powershell. For now we've only seen the syntax of the language, but in the next class we'll jump right into the action with harder examples and harder algorithms. We left you a couple of exercises with this documentation, please take the time to do them. As a last thing we leave you with an advice.

Powershell, as any other programming/scripting language, requires time and dedication to master, do not get discouraged by a script that does not work as you intended or by an Exception you don't understand, use the best tool that powershell has to offer, its community. There are a lot of people that will help you no matter how dumb the issue may seem, you will find someone that has that issue and is looking for an answer too, or someone who had that issue and resolved it. Community links will be provided with the exercises.