BE A GOOD POWERSHELL CITIZEN

Paul Broadwith

@pauby   pauby.com   github.com/pauby

ABOUT ME

- Paul Broadwith
  - Freelancer since 2001
  - 25 years in IT in the defence, government, financial services and nuclear industry sectors;
- Contact Me

🏠 https://pauby.com

@pauby

https://github.com/pauby

https://www.linkedin.com/in/paulbroadwith/

WHAT MAKES A GOOD CITIZEN

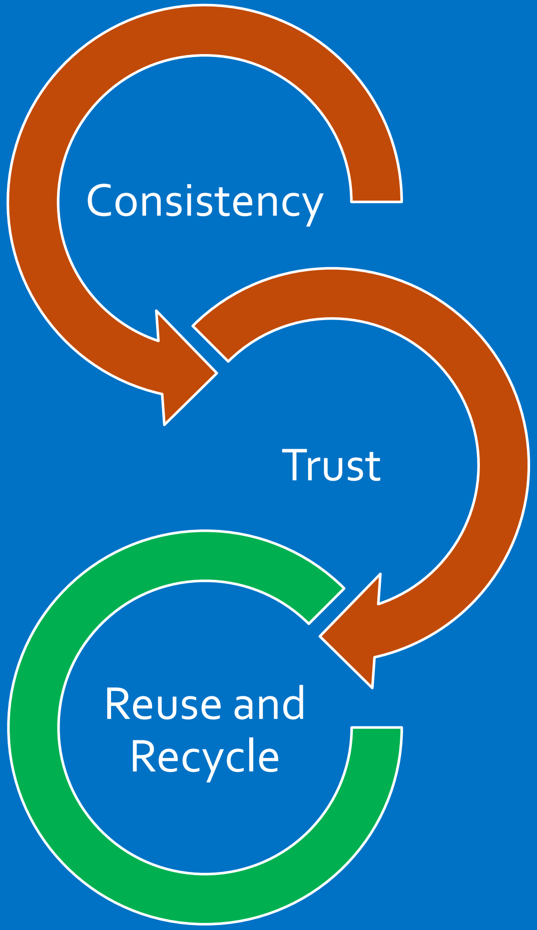
- Every group, community or organisation has:
  - Rules
  - Standards
  - Best practice
  - Guidelines
- Following the ‘rules’ makes you a good citizen
- Makes it easier to interact and communicate with peers and colleagues

10 DO'S AND DON'T'S OF POWERSHELL

1. Develop a style and stick to it
2. Use Advanced Functions
3. Leverage built-in validation
4. Name your things well
5. Filter left, format right
6. Sprinkle comments
7. Avoid technical debt, write help now
8. Use the pipeline and objects
9. Don't pollute the users session
10. Go green with your code – Reduce, Reuse and Recycle

IMPORTANCE OF THE DO'S AND DON'T'S OF POWERSHELL

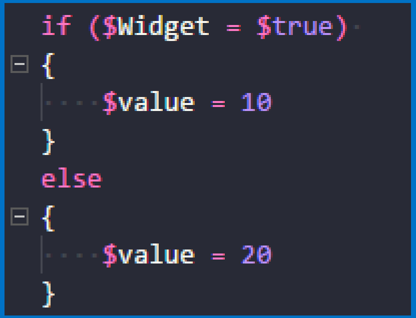
- If you leave, get sick or get hit by a bus
- Peer review easier
- Post code online such as PowerShell Gallery or Github
- **Consistency** leads to **Trust** leads to **Reuse and Recycle**



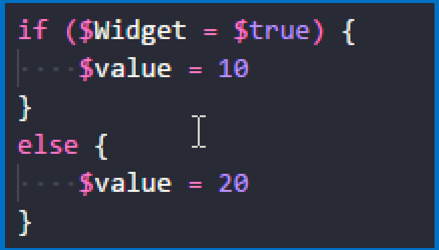
DEVELOP A STYLE AND STICK TO IT

DEVELOP A STYLE AND STICK TO IT

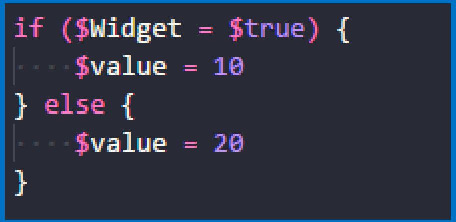
- Choose a bracing style, naming style, help style and comment style:



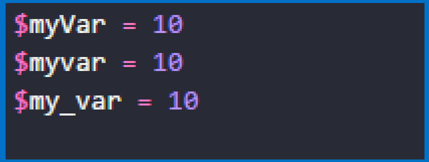
```
if ($Widget = $true)
{
    $value = 10
}
else
{
    $value = 20
}
```



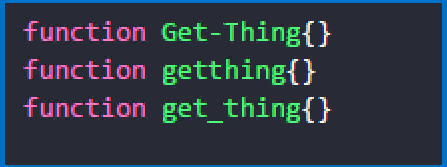
```
if ($Widget = $true) {
    $value = 10
}
else {
    $value = 20
}
```



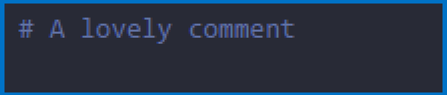
```
if ($Widget = $true) {
    $value = 10
} else {
    $value = 20
}
```



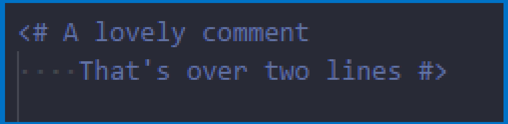
```
$myVar = 10
$myvar = 10
$my_var = 10
```



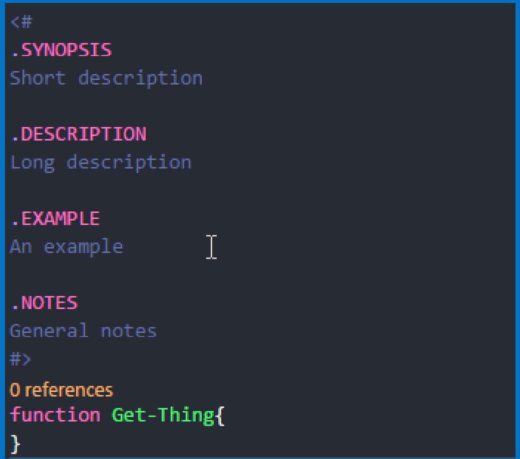
```
function Get-Thing{}
function getthing{}
function get_thing{}
```



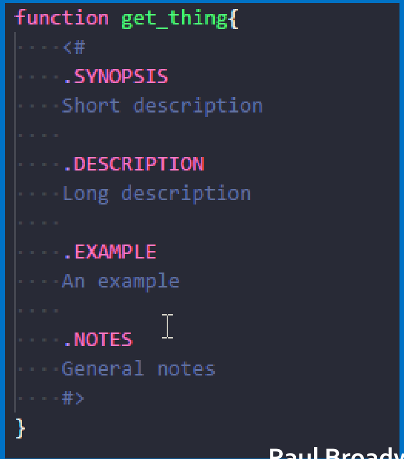
```
# A lovely comment
```



```
<# A lovely comment
    That's over two lines #>
```



```
<#
.SYNOPSIS
Short description

.DESCRIPTION
Long description

.EXAMPLE
An example

.NOTES
General notes
#>
0 references
function Get-Thing{
}
```



```
function get_thing{
    <#
    .SYNOPSIS
    Short description

    .DESCRIPTION
    Long description

    .EXAMPLE
    An example

    .NOTES
    General notes
    #>
}
```

- Whatever you choose, BE CONSISTENT!

Paul Broadwith @pauby

DEVELOP A STYLE AND STICK TO IT

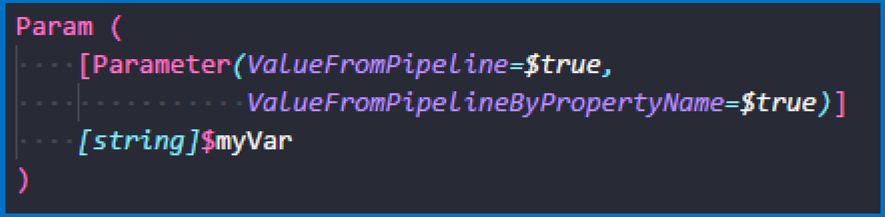
- Don't:
  - Use Hungarian Notation – dynamically typed language
  - Use aliases in code – they may not be available or change
  - Rely on positional parameters in code – they may change
- Do:
  - Use full cmdlet, function and parameter names in your code

USE ADVANCED FUNCTIONS

Paul Broadwith @pauby

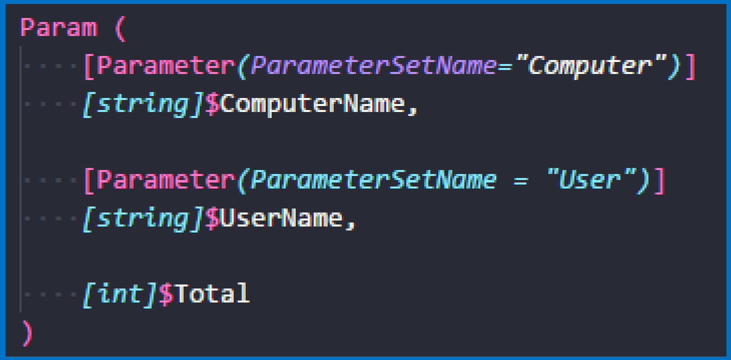
WHY USE ADVANCED FUNCTIONS

- Allows your function to accept **`-Verbose`**, **`-Debug`**, **`-WhatIf`**, **`-Confirm`**, **`-ErrorAction`** and others.
- Access to the pipeline



```
Param (
    [Parameter(ValueFromPipeline=$true,
               ValueFromPipelineByPropertyName=$true)]
    [string]$myVar
)
```

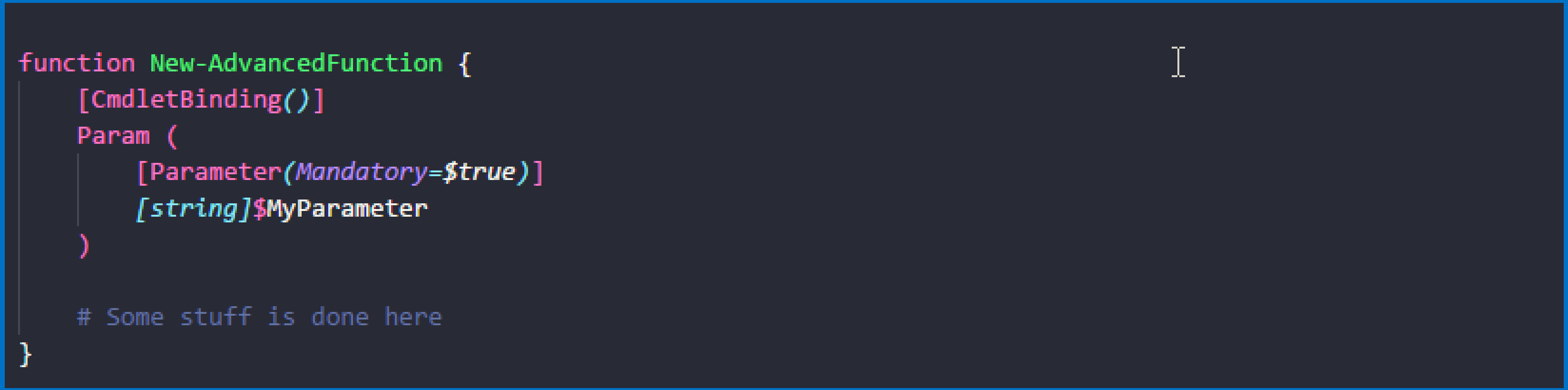
- Parameter Sets



```
Param (
    [Parameter(ParameterSetName="Computer")]
    [string]$ComputerName,

    [Parameter(ParameterSetName = "User")]
    [string]$UserName,

    [int]$Total
)
```

JUST DO IT!

- Define advanced function using **`[CmdletBinding()]`**



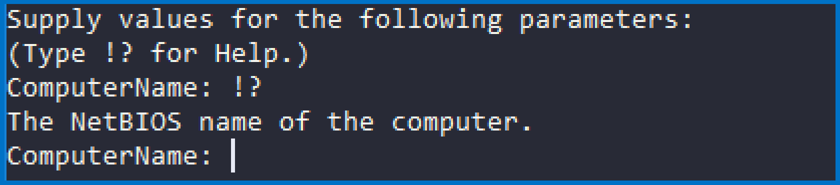
```
function New-AdvancedFunction {
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory=$true)]
        [string]$MyParameter
    )

    # Some stuff is done here
}
```

- Use **`man about_functions_advanced`**

LEVERAGE BUILT-IN VALIDATION

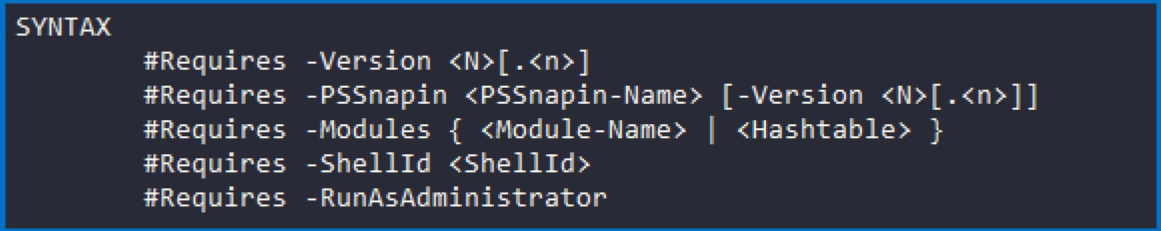
LEVERAGE BUILT-IN VALIDATION

- **Parameter** Attributes
  - **Mandatory** – prompts if parameter is missing.
  - **HelpMessage** –what is required



```
Supply values for the following parameters:
(Type !? for Help.)
ComputerName: !?
The NetBIOS name of the computer.
ComputerName:
```

- **#Requires** statement
  - States code pre-requisites

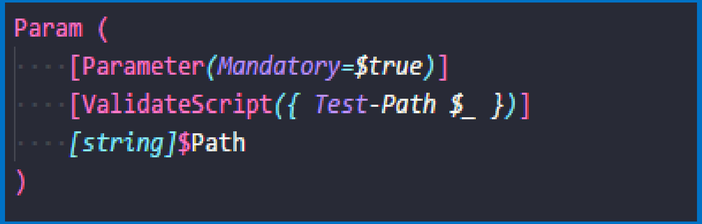


```
SYNTAX
        #Requires -Version <N>[.<n>]
        #Requires -PSSnapin <PSSnapin-Name> [-Version <N>[.<n>]]
        #Requires -Modules { <Module-Name> | <Hashtable> }
        #Requires -ShellId <ShellId>
        #Requires -RunAsAdministrator
```

- **Set-StrictMode** statement
  - Generates a terminating error when basic best-practice coding rules are violated

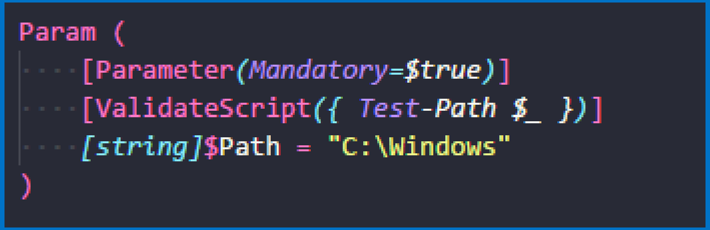
LEVERAGE BUILT-IN VALIDATION

- Validation attributes
  - `[ValidateCount(min, max)]`
  - `[ValidateLength(min, max)]`
  - `[ValidatePattern(<REGEX>)]`
  - `[ValidateScript({<SCRIPTBLOCK>})]`



```
Param (
    [Parameter(Mandatory=$true)]
    [ValidateScript({ Test-Path $_ })]
    [string]$Path
)
```

- **Assign defaults to parameters**



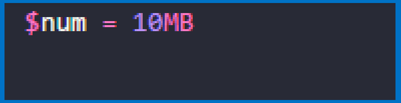
```
Param (
    [Parameter(Mandatory=$true)]
    [ValidateScript({ Test-Path $_ })]
    [string]$Path = "C:\Windows"
)
```

NAME YOUR THINGS WELL

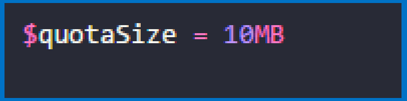
Paul Broadwith @pauby

NAME YOUR THINGS WELL

- Use common parameter names
  - **`-Path`**, **`-Computername`**, **`-Destination`**
- Use singular naming
  - **`Get-Item`**, **`Get-ADUser`**, **`Add-AppxPackage`**
- **Use descriptive names for variables, functions, parameters and modules:**



```
$num = 10MB
```



```
$quotaSize = 10MB
```

FILTER LEFT, FORMAT RIGHT

FILTER LEFT, FORMAT RIGHT

- Filter at the source

```
Get-AdUser -Filter ( samAccountName -like "98*" )
```

- Not afterwards

```
Get-AdUser -Filter * | Where-Object { $_.samAccountName -like "98*"}
```

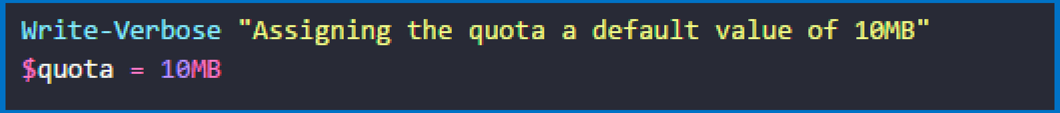
- Format before output

```
Get-AdUser -Filter * | Where-Object { $_.samAccountName -like "98*"} |
    Format-Table samAccontName,Name,Department |
    Export-Csv C:\ADUsers.txt
```

SPRINKLE COMMENTS

COMMENTING YOUR CODE

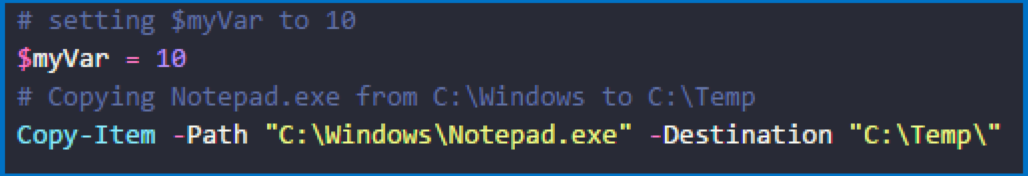
- Top-down linear code almost comments itself
- Use Write-Verbose to comment your code
  - Displayed on the host with the **–Verbose** parameter
  - Describes your code as you go
- Comment for somebody else
  - You know what it does and how it does it, the rest of the world does not!



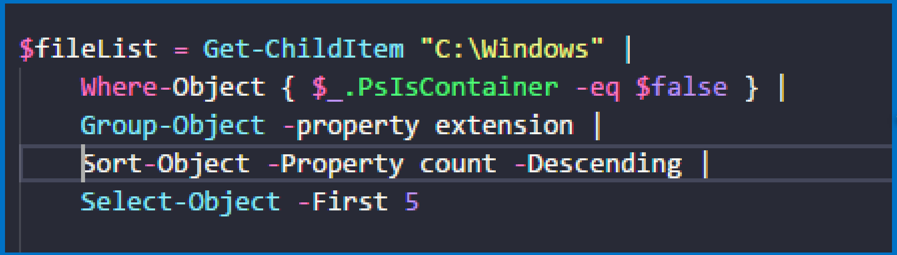
```
Write-Verbose "Assigning the quota a default value of 10MB"
$quota = 10MB
```

COMMENTING NIRVANA

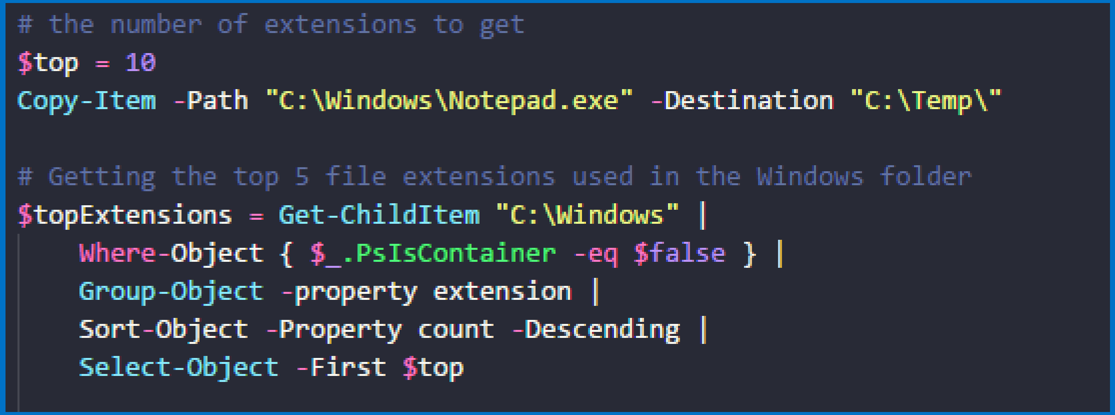
- Comments should not explain the obvious



```
# setting $myVar to 10
$myVar = 10
# Copying Notepad.exe from C:\Windows to C:\Temp
Copy-Item -Path "C:\Windows\Notepad.exe" -Destination "C:\Temp\"
```



```
$fileList = Get-ChildItem "C:\Windows" |
    Where-Object { $_.PsIsContainer -eq $false } |
    Group-Object -property extension |
    Sort-Object -Property count -Descending |
    Select-Object -First 5
```



```
# the number of extensions to get
$top = 10
Copy-Item -Path "C:\Windows\Notepad.exe" -Destination "C:\Temp\"

# Getting the top 5 file extensions used in the Windows folder
$topExtensions = Get-ChildItem "C:\Windows" |
    Where-Object { $_.PsIsContainer -eq $false } |
    Group-Object -property extension |
    Sort-Object -Property count -Descending |
    Select-Object -First $top
```

- Comments should be used to explain the not-so-obvious

Paul Broadwith @pauby

AVOID TECHNICAL DEBT – WRITE HELP NOW

HELP ON HELP

- Says what your code does
  - .SYNOPSIS
  - .DESCRIPTION
- Says what parameters are available, how and required
  - .PARAMETER
- Gives a demo of how to use the code
  - .EXAMPLE
- Add help to each function you write as you go along – don't pretend you will do it later!

WRITING HELP

- As a minim
  - .S
    -

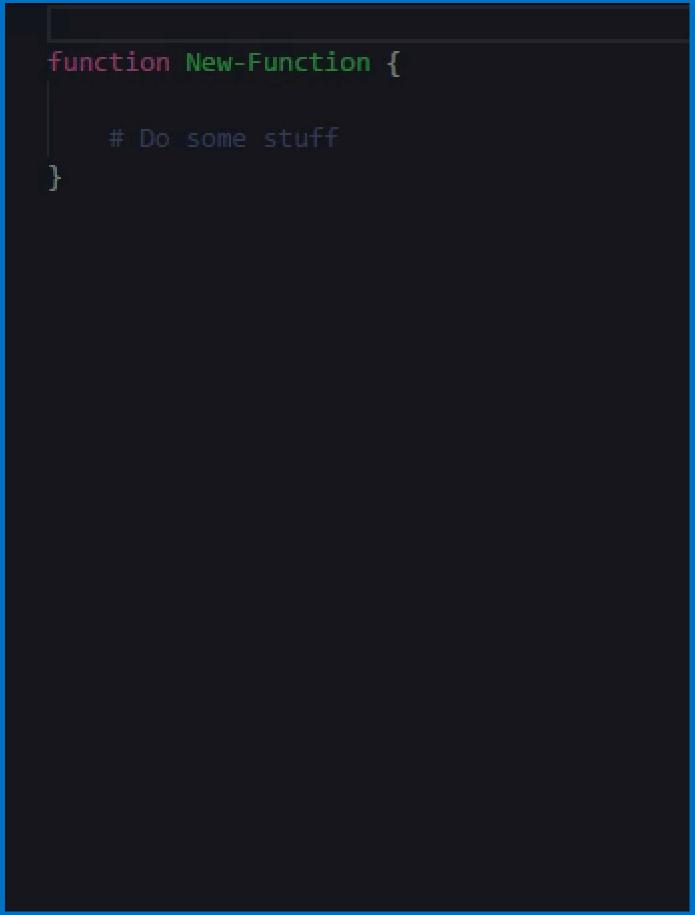
ur code

- Us
 (with Pester)

Use <# before or within a function

QUICKLY ADD HELP TO YOUR CODE

Use Control-J and select *Cmdlet (advanced function)*



```
function New-Function {

    # Do some stuff
}
```

USE THE PIPELINE AND OBJECTS

AVOID WRITE-HOST, EXCEPT WHEN YOU CAN'T

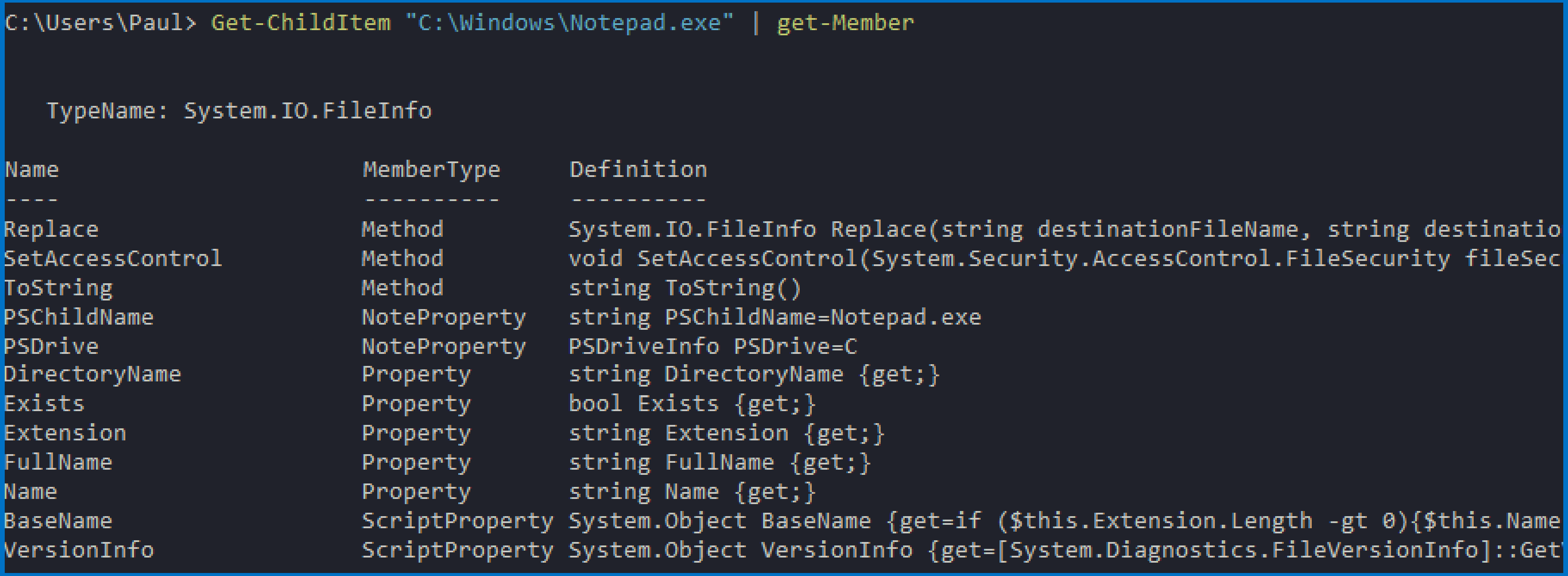
- Write-Host output cannot be captured
- Allow the user to choose if they want to see your **`“I’m doing this”`** messages with **`Write-Verbose`**
- Use `Write-Debug` to display debugging information such as contents of viarables
- Use **`Write-Warning`** or **`Write-Error`** to notify

AVOID WRITE-HOST, EXCEPT WHEN YOU CAN'T

- Write-Host is the only cmdlet to display coloured text
- It allows formatted(ish) text with **`-NoNewLine`**
- It's easy and quick to use
- The user will always be shown it
- The user does not have to do anything
  - No need to add **`–Verbose`** or **`–Debug`** parameters

OBJECTS

- Anatomy of an object:
  - Properties
    - Size
    - Length
    - Name
  - Methods
    - Trim()
    - ToString()

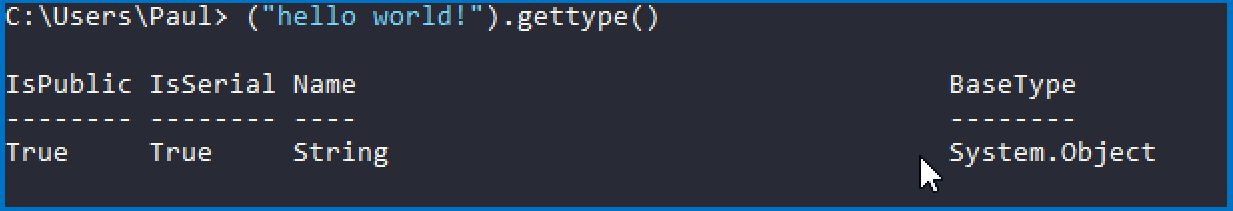


```
C:\Users\Paul> Get-ChildItem "C:\Windows\Notepad.exe" | get-Member


   TypeName: System.IO.FileInfo

Name                          MemberType     Definition
----                          ----------     ----------
Replace                       Method         System.IO.FileInfo Replace(string destinationFileName, string destinatio
SetAccessControl              Method         void SetAccessControl(System.Security.AccessControl.FileSecurity fileSec
ToString                      Method         string ToString()
PSChildName                   NoteProperty   string PSChildName=Notepad.exe
PSDrive                       NoteProperty   PSDriveInfo PSDrive=C
DirectoryName                 Property       string DirectoryName {get;}
Exists                        Property       bool Exists {get;}
Extension                     Property       string Extension {get;}
FullName                      Property       string FullName {get;}
Name                          Property       string Name {get;}
BaseName                      ScriptProperty System.Object BaseName {get=if ($this.Extension.Length -gt 0){$this.Name
VersionInfo                   ScriptProperty System.Object VersionInfo {get=[System.Diagnostics.FileVersionInfo]::Get
```

Paul Broadwith @pauby

OBJECTS

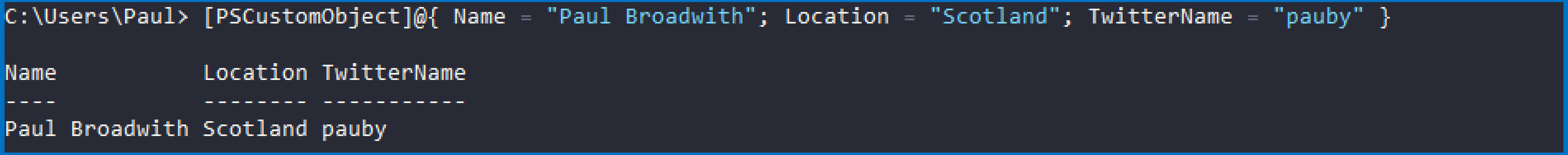
- EVERYTHING in PowerShell is an object



```
C:\Users\Paul> ("hello world!").gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

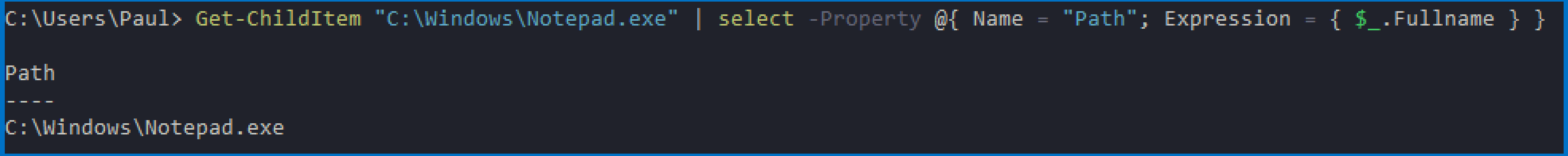
CREATING YOUR OWN OBJECTS

- Create your own objects for output



```
C:\Users\Paul> [PSCustomObject]@{ Name = "Paul Broadwith"; Location = "Scotland"; TwitterName = "pauby" }

Name           Location TwitterName
----           -------- -----------
Paul Broadwith Scotland pauby
```

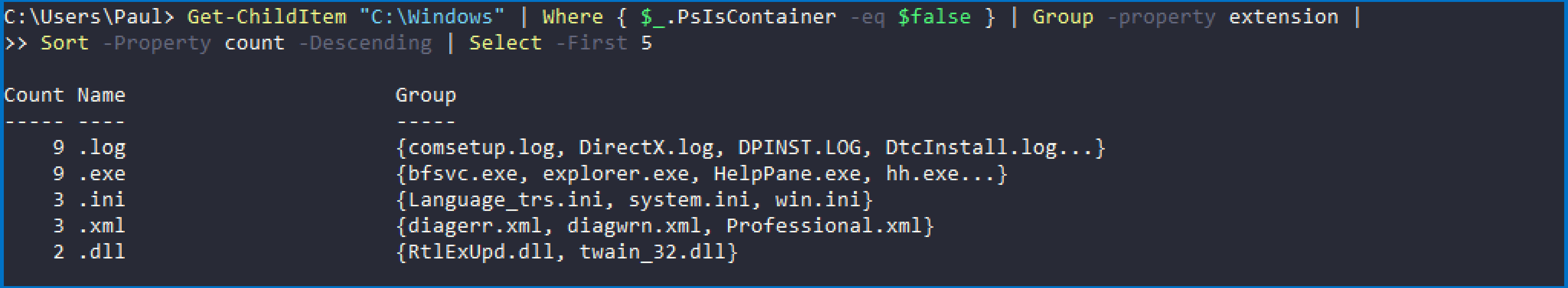
- Bend the output of other cmdlets to your will!



```
C:\Users\Paul> Get-ChildItem "C:\Windows\Notepad.exe" | select -Property @{ Name = "Path"; Expression = { $_.Fullname } }

Path
----
C:\Windows\Notepad.exe
```

USE THE PIPELINE

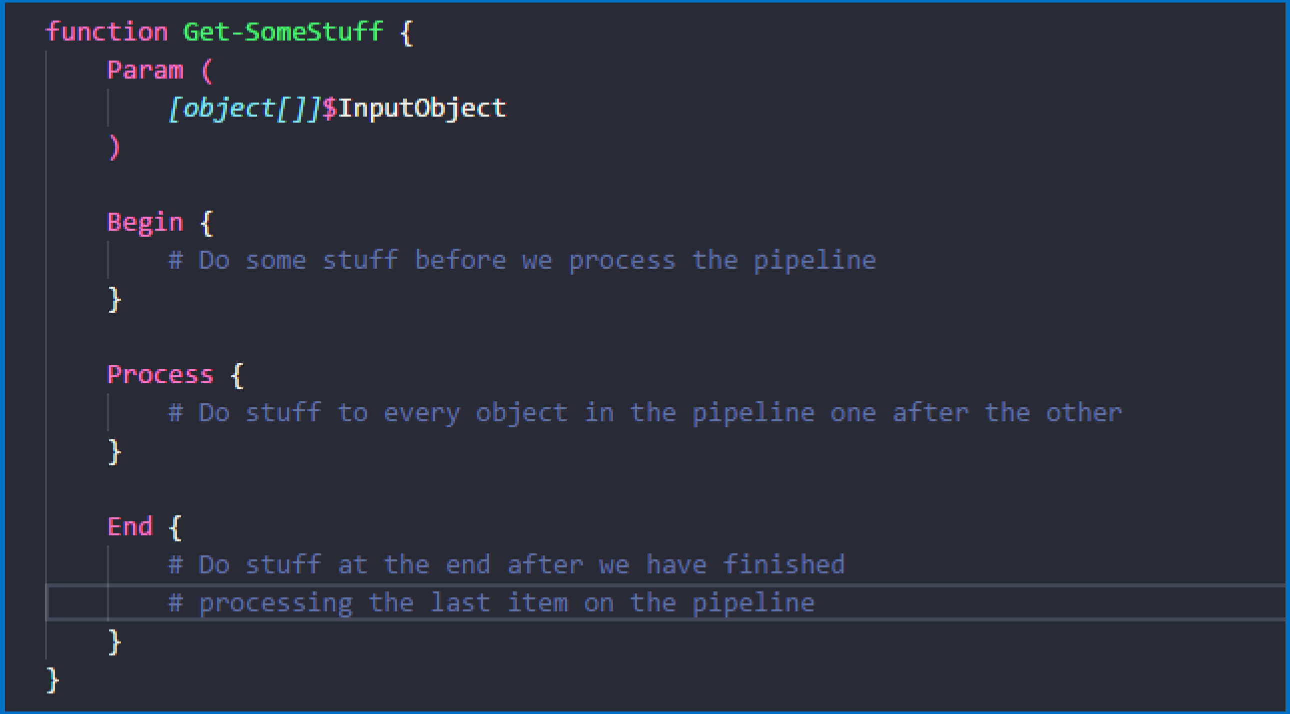
- Most cmdlets use the pipeline
- Code to use the pipeline
- Allows cmdlets and functions to be chained together



```
C:\Users\Paul> Get-ChildItem "C:\Windows" | Where { $_.PsIsContainer -eq $false } | Group -property extension |
>> Sort -Property count -Descending | Select -First 5

Count Name                      Group
----- ----                      -----
    9 .log                      {comsetup.log, DirectX.log, DPINST.LOG, DtcInstall.log...}
    9 .exe                      {bfsvc.exe, explorer.exe, HelpPane.exe, hh.exe...}
    3 .ini                      {Language_trs.ini, system.ini, win.ini}
    3 .xml                      {diagerr.xml, diagwrn.xml, Professional.xml}
    2 .dll                      {RtlExUpd.dll, twain_32.dll}
```

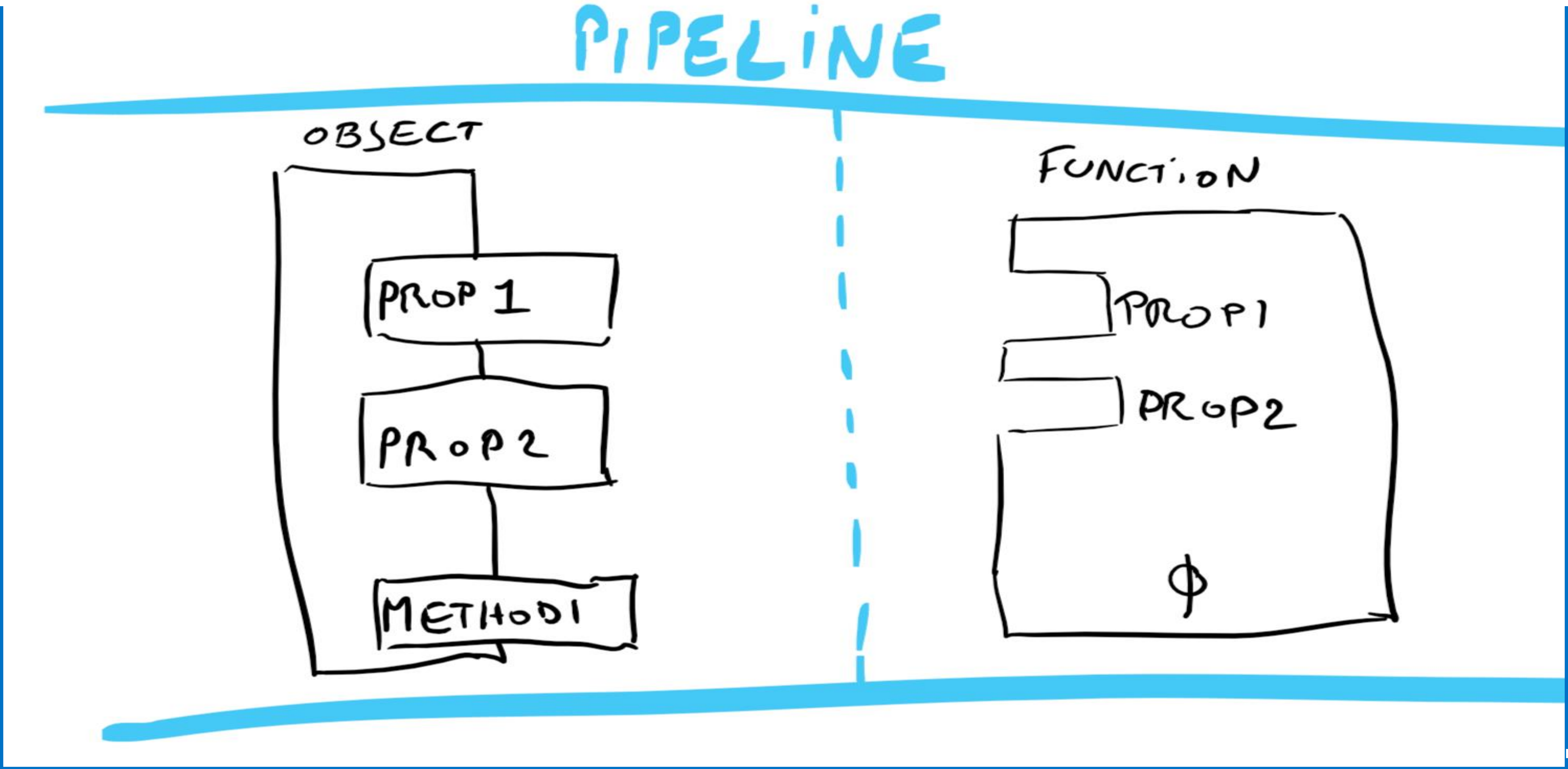
USE THE PIPELINE

- Code to use the pipeline



```
function Get-SomeStuff {
    Param (
        [object[]]$InputObject
    )

    Begin {
        # Do some stuff before we process the pipeline
    }

    Process {
        # Do stuff to every object in the pipeline one after the other
    }

    End {
        # Do stuff at the end after we have finished
        # processing the last item on the pipeline
    }
}
```

IT'S ALL ABOUT THE PIPELINE



DON'T POLLUTE THE USERS SESSION

UNDERSTANDING SCOPE

Scope	What's in it
Global	Everything created when PowerShell starts; Everything created at the console;
Script	Created when a script runs and only commands in the script run in this scope;
Local	Current scope and can be any scope;
Private	Cannot be seen outside of the current scope

Numbered Scope	Relative scopes;
	0 is current scope;
	1 is parent scope
	2 is parent's parent scope…

DON'T POLLUTE THE USERS SESSION

- Don't use **`$ErrorActionPreference`**
- Don't clear the screen buffer using **`CLS`**!
- Use **`$Script:`** and not **`$Global:`** for creating and referencing 'script global' variables
- Save any changes you have to make and restore them when complete

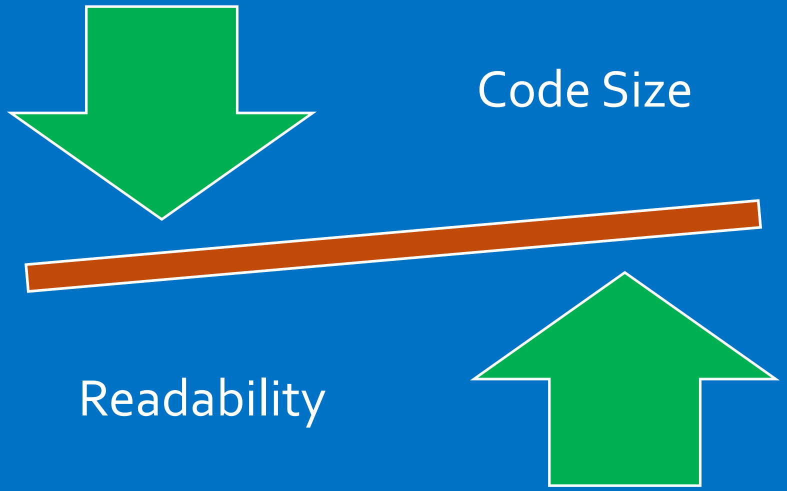
GO GREEN WITH YOUR CODE

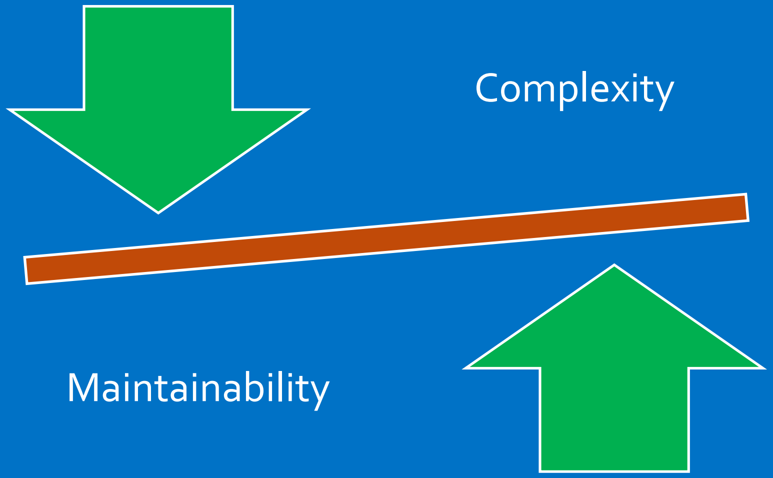
Paul Broadwith @pauby

SAVE PLANET CODE – GO GREEN

Reduce

- Code size reduces
- Readability increases



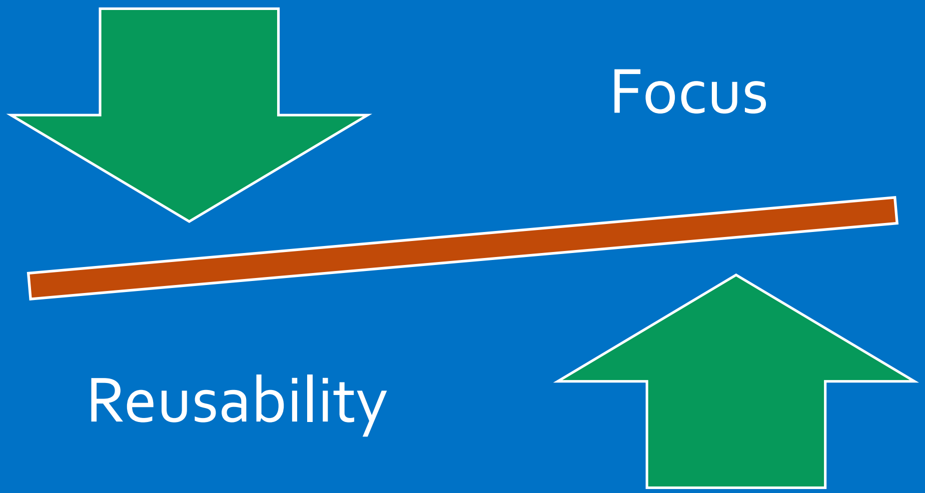


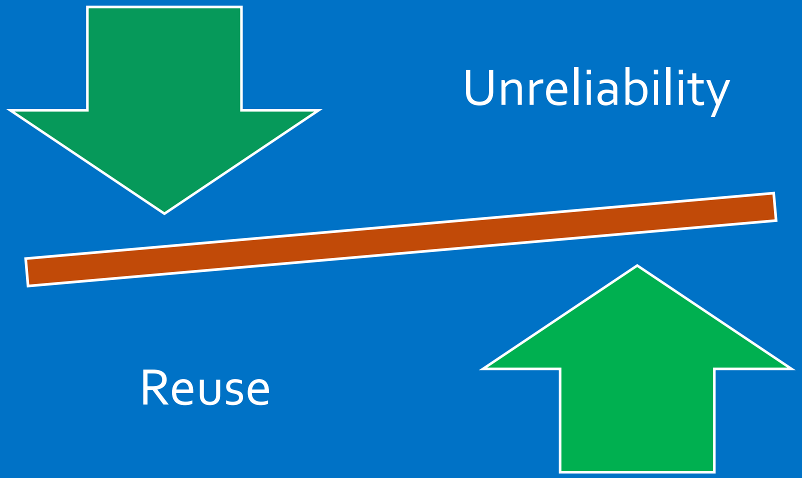
- Complexity reduces
- Maintainability increases

SAVE PLANET CODE – GO GREEN

Reuse

- Focus narrows
- Reusability increases





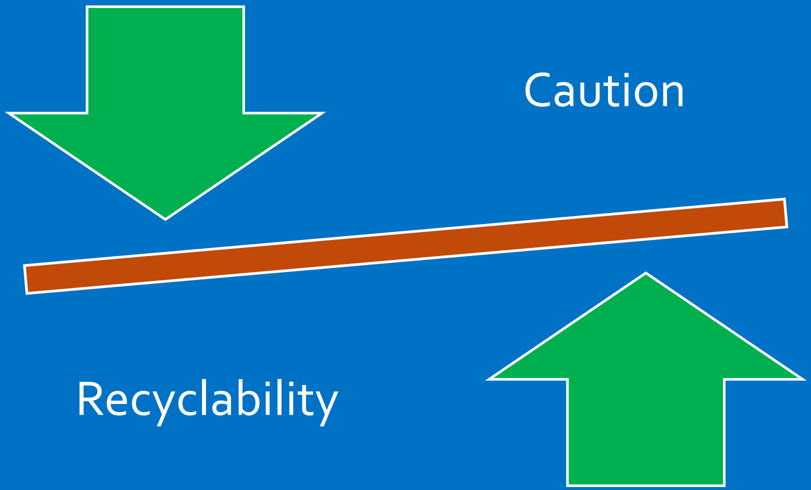
- Reuse increases
- Potential for bugs & unreliability decreases

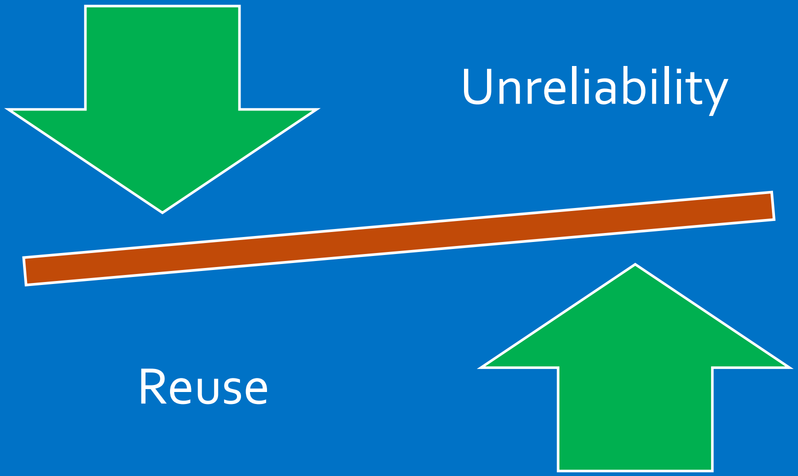
SAVE PLANET CODE – GO GREEN

Recycle

Steal from the best, write the rest. – Ed Wilson, The Scripting Guy

- Caution decreases
- Recyclability increases





- Dependencies decrease
- Recyclability increases

QUESTIONS?

RESOURCES

- PowerShell Practice & Style Guide
  - https://github.com/PoshCode/PowerShellPracticeAndStyle

THANK YOU!

Paul Broadwith

@pauby

pauby.com

github.com/pauby