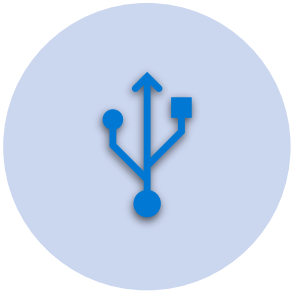
Microsoft

Functions - Advanced

Function Overview

Reusable block of PowerShell code

Reduces size of code and increases reliability

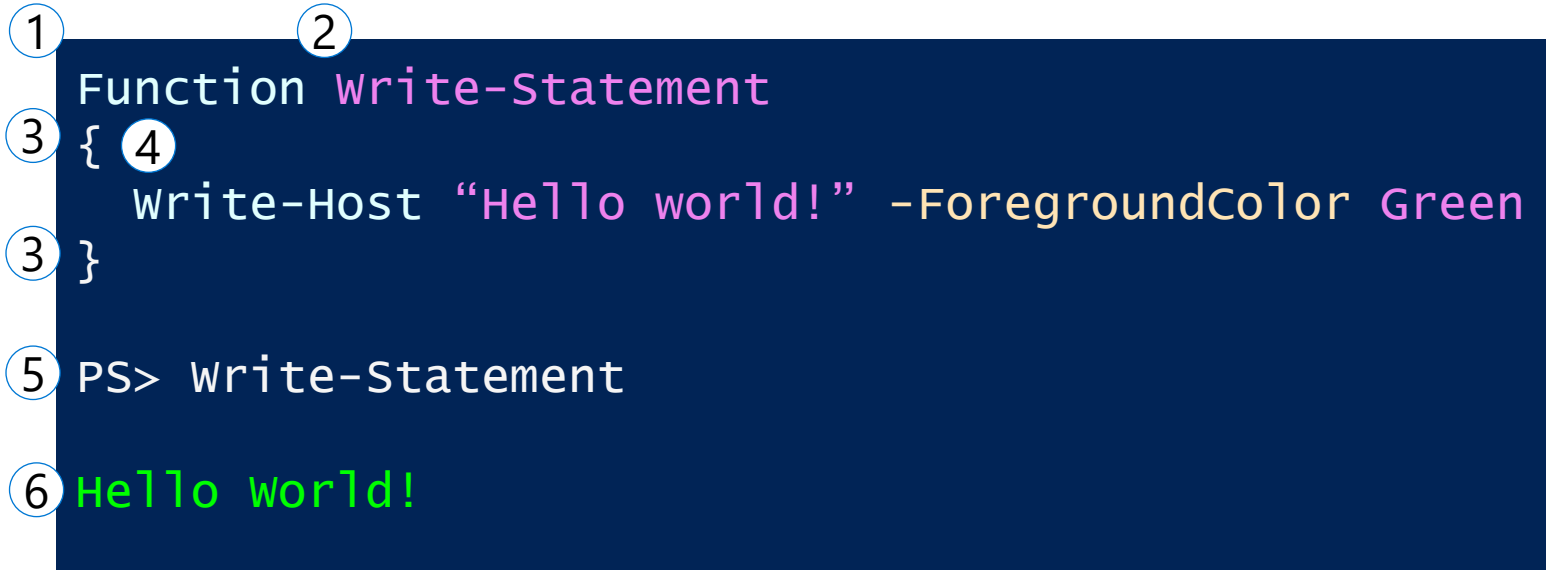


Can accept **parameter** values and return output

Advanced Functions behave like **Cmdlets**, including help content

What Does a Function Look Like?

1. Function Keyword
2. Function Name
3. Curly Brace Pair
4. PowerShell Commands
5. Call the Function
6. Function output



Creating a Function

Multiple commands can be contained within a function

Allows for large code blocks to be reused – no need to copy and paste

Maintains consistency between repeated uses of code when edited

```
Function Write-ServiceStatus
{
  $SVC = Get-Service -Name WinRM
  $Name = $SVC.DisplayName
  $Status = $SVC.Status
  Write-Host "The Service $Name is currently $Status" -ForegroundColor Green
}

PS> Write-ServiceStatus
The Service Windows Remote Management (WS-Management) is currently Stopped
```

Add Parameters to a Function

1. Function Keyword
2. Function Name
3. Curly Brace Pair
4. Parameter statement
5. PowerShell Commands
6. Call the Function
7. Function output

```
Function Write-Statement
{
  Param($Statement)
  Write-Host $Statement -ForegroundColor Green
}

PS> Write-Statement -Statement "Hello World!"

Hello World!
```

Default Parameter Values

Can assign a value like any other variable

Function parameters will override the default value

```
Function Write-Statement
{
  Param($Statement = "Good morning!")
  Write-Host $Statement -ForegroundColor Green
}

PS> Write-Statement

Good morning!
```

Adding Multiple Parameters

Functions can accept multiple parameters separated by commas

Supports line breaks between parameters

```
Function Write-ServiceStatus
{
  Param ($Service,
         $Color = "Green")
  $SVC = Get-Service -Name $Service
  $Name = $SVC.DisplayName
  $Status = $SVC.Status
  Write-Host "The Service $Name is currently $Status" -ForegroundColor $Color
}

PS> Write-ServiceStatus  -Service WinRM -Color Yellow
The Service Windows Remote Management (WS-Management) is currently Stopped
```

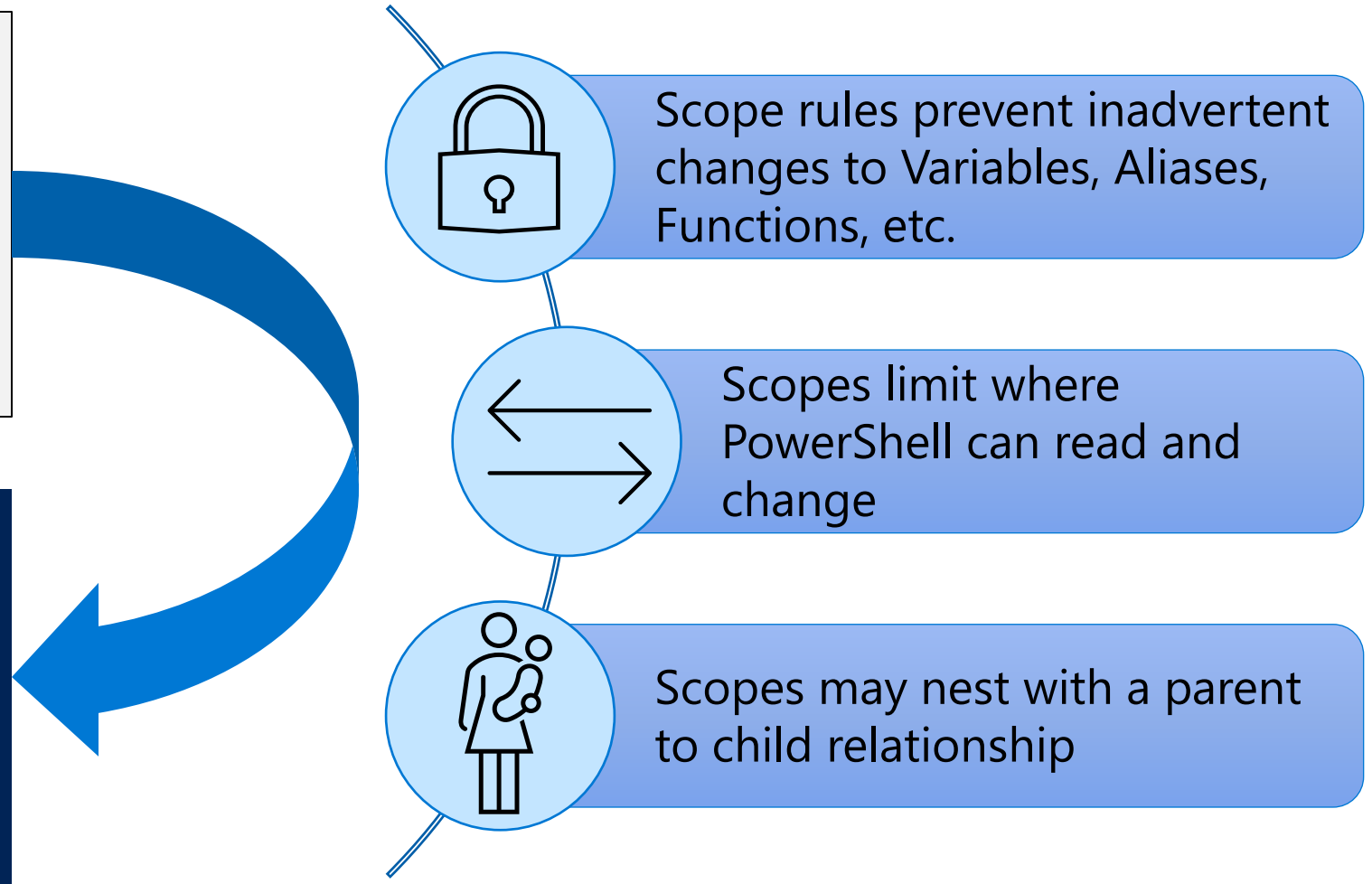
Questions?

Scopes

What is a scope?

```
Function Show-Scope
{
    $Service = “ALG”
    $Service
}
```

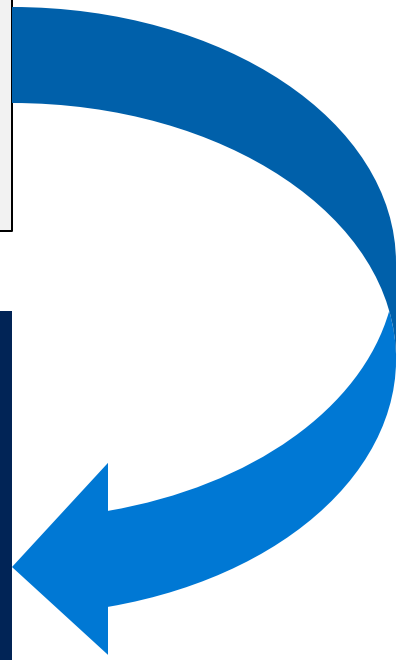
```
PS> $Service = "WinRM"
PS> Show-Scope
ALG
PS> $Service
WinRM
```



Dot-Source Notation

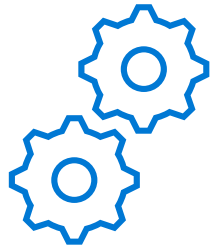
```
Function Show-Scope
{
    $Service = "ALG"
    $Service
}
```

```
PS> $Service = "WinRM"
PS> . Show-Scope
ALG
PS> $Service
ALG
```



Dot-source notation does not create a child scope

This creates objects in the current scope



Useful for loading toolbox scripts

Scope Modifiers

Global	• Highest level scope, usually the host • Avoid using as it will overwrite existing values
Script	• Nearest script scope • Contains variables to just your script
Local	• Current scope (Default)
Private	• Current scope, unreadable by child scopes • Used for advanced tool creation

```
Function Global:Hello {Write-Host "Hello World!"}
$Script:MyVar = "Good morning!"
```

Parent – Child Scope Relationship

A child scope can view a parent scopes values

```
Function Show-Scope
{
    $Service
    $Service = “ALG”
    $Service
}
```

```
PS> $Service = "WinRM"
PS> Show-Scope
WinRM
ALG
PS> $Service
WinRM
```

The value in the parent scope is unchanged

Demonstration: Scopes

Parameter Overview

- Function/script parameters can be named, positional, or dynamic

- Can be defined in three ways:
  - Param() keyword in body
  - Following function name before body
  - $args automatic variable (positional only)

Parameters – No Param() statement

- Can appear in parenthesis () prior to the function body

- [CmdletBinding()] attribute not available
  - Discussed later

```
PS C:\> function <Name> ($p1,$p2,$pn)
{
    <PowerShell code>
}
```

Parameters – Param() statement

- Full set of advanced function parameter features available when using Param() within body

```
function <Name> {
    param ($p1,$p2,$pn)
}
```

- Set default values

```
function <Name> {
    param ($p1 ="Value1",$p2 ="Value2",$pn ="ValueN")
}
```

- Strongly typed parameters

```
function <Name> {
    param ([int]$p1 ="42", [string]$p2 ="Value2")
}
```

Parameter Binding

- PowerShell passes undefined parameters to $args
  - Includes both parameter name and argument value

- $PSBoundParameters returns hash table containing bound values to known parameters

Parameters – Unnamed, positional

- Automatic variable - $args
- Array of undeclared parameter values passed to function, script, or script block
- Least-preferred technique

```
function Use-Args {$args[0],$args[1]}

Use-Args Hello World
```

$PSBoundParameters

- Contains a dictionary of the parameters that are passed to a script or function and their current values. This variable has a value only in a scope where parameters are declared, such as a script or function. You can use it to display or change the current values of parameters or to pass parameter values to another script or function.

```
function Get-PSBoundParameters {
       param (
           [int]$param1,
           [string]$param2
       )
       Write-Output "Parameters accepted are $param1 and $param2“
       Write-Output $PSBoundParameters
}
```

Positional Arguments

- Great for simple functions
- Automatically implemented
- Order matters
- No built-in type validation
- Can be confusing with many arguments

```
PS C:\> Get-SystemUptime SV-001
```

```
PS C:\> Add-Numbers 45 76 89 23 74
```

```
PS C:\> New-DataRecord jdoe John Doe NJ ‘Red Bank’ US
```

Named Parameters

- Implement with the function declaration
- Implement with the Param() block
- Order doesn't matter
- Many validation options
- Additional parameters in $args

```
PS C:\> Get-SystemUptime -Computername SV-001
```

```
PS C:\> Add-Numbers -Values 45,76,89,23,74
```

```
PS C:\> New-DataRecord -Username jdoe –Givenname John `
    –Surname Doe NJ –City 'Red Bank' US
```

Questions?

Demonstration: Parameters Basics

Switch Parameter

- Parameters with no parameter value
- To create a switch parameter in a function use the [Switch] type
- False by default, True if present
- Syntax

```
Param ([Switch]<ParameterName>)


function SwitchExample {
    Param([switch]$state)
    if ($state) {"Switch is on"} else {"Switch is off"}
}
```

Switch Parameter (Continued)

Example

```
function SwitchExample {
    Param([switch]$state)
    if ($state) {"Switch is on"} else {"Switch is off"}
}
```

```
PS C:\> SwitchExample
Switch is Off

PS C:\> SwitchExample -state
Switch is On

PS C:\> SwitchExample -state:$false
Switch is Off

PS C:\> SwitchExample -state:$true
Switch is on
```

Switch parameter design considerations

- Switch parameters should not be given default values. They should always default to false.
- Switch parameters are excluded from positional parameters by default.
- Switch parameters should not be mandatory.

Demonstration

Parameter Basics

Questions?

The Parameter Attribute

[Parameter] Attribute

- Parameter attribute allows us to describe the parameter that we are declaring
- We have several attributes that we can 'attach' to the parameter

```
function Get-ParameterAttribute {
    Param(
        [Parameter]
        [string[]]
        $ComputerName
    )
    Write-Output "Computer name passed is: $ComputerName"
}
```

Mandatory Parameter attribute

- The Parameter attribute is used to declare the attributes of function parameters.
- The Parameter attribute is optional
- The Parameter attribute has arguments that define the characteristics of the parameter, such as whether the parameter is mandatory or optional.
- Syntax

```
function Get-MandatoryParameter {
    Param(
        [Parameter(Mandatory)]
        [string[]]
        $ComputerName
    )
}
```

Positional parameter

- The Position argument determines whether the parameter name is required when the parameter is used in a command.
- By default, all function parameters are positional (can be disabled). PowerShell assigns position numbers to parameters in the order in which the parameters are declared in the function.
- Syntax

```
function Get-PositionalParameter {
    Param(
        [Parameter(Position=0)]
        [string[]]
        $ComputerName
    )
    Write-Output "Computer name passed is: $ComputerName"
}
```

ParameterSetName argument

- The ParameterSetName argument specifies the parameter set to which a parameter belongs.
- If no parameter set is specified, the parameter belongs to all the parameter sets defined by the function.
- Therefore, to be unique, each parameter set must have at least one parameter that isn't a member of any other parameter set.

```
Param(
        [Parameter(Mandatory,
        ParameterSetName="Computer")]
        [string[]]
        $ComputerName,
        [Parameter(Mandatory,
        ParameterSetName="User")]
        [string[]]
        $UserName,
        [Parameter()]
        [switch]
        $Summary
    )
```

ValueFromPipeline argument (ByValue)

- The ValueFromPipeline argument indicates that the parameter accepts input from a pipeline object. Specify this argument if the function accepts the entire object, not just a property of the object.
- Syntax

```
Function Get-ValueFromPipeline{
    Param(
        [Parameter(Mandatory,
        ValueFromPipeline)]
        [string[]]
        $ComputerName
    )
    Write-Output "Computer name passed is: $ComputerName"
}
'MyComputer' | Get-ValueFromPipeline
```

ValueFromPipelineByPropertyName argument

- The **ValueFromPipelineByPropertyName** argument indicates that the parameter accepts input from a property of a pipeline object. The object property must have the same name or alias as the parameter.

```
function Get-ValueFromPipelineByPropertyName{
    param (
        [Parameter(Mandatory,
        ValueFromPipelineByPropertyName)]
        [string[]]
        $ComputerName
    )
    Write-Output "Computer name passed is: $ComputerName"
}
```

HelpMessage argument

- The HelpMessage argument specifies a string that contains a brief description of the parameter or its value.
- PowerShell displays this message in the prompt that appears when a mandatory parameter value is missing from a command.
- This argument has **no effect on optional parameters**.

```
Function Get-HelpMessage {
    Param(
        [Parameter(Mandatory,
            HelpMessage = 'Enter computer name')]
        [string]
        $ComputerName
    )
}
```

Alias argument

- The Alias attribute establishes an alternate name for the parameter. There's no limit to the number of aliases that you can assign to a parameter.

```
function Get-AliasParameter{
    param (
        [Parameter(Mandatory,
        ValueFromPipelineByPropertyName)]
        [Alias('cn','MachineName')]
        [string[]]
        $ComputerName
    )
}
```

Demonstration

Parameter Attribute

Questions?
Parameter Attribute

Advanced attributes & Validators

AllowNull / AllowEmptyString / AllowEmptyCollection attribute

- These attributes allow you to allows the value of a mandatory parameter to be an empty $null to a parameter

```
function Get-AllowNull{
    Param(
        [Parameter(Mandatory)]
        [AllowNull()]
        [hashtable]
        $ComputerInfo
    )
    Write-Output $ComputerInfo
}
Get-AllowNull -ComputerInfo $null
Get-AllowNull -ComputerInfo @{'value' = 1}
```

ValidateNotNull / ValidateNotNullOrEmpty attribute

- The ValidateNotNull attribute specifies that the parameter value can't be $null. PowerShell generates an error if the parameter value is $null.
- The ValidateNotNullOrEmpty attribute specifies that the parameter value can't be $null and can't be an empty string (""). PowerShell generates an error if the parameter is used in a function call, but its value is $null, an empty string (""), or an empty array @().

```
function Get-ValidateNotNullOrEmpty {
    Param(
        [Parameter(Mandatory)]
        [ValidateNotNullOrEmpty()]
        [string[]]
        $UserName
    )
    Write-Output $username
}
Get-ValidateNotNullOrEmpty -UserName 'Gadi’
Get-ValidateNotNullOrEmpty -UserName ’’
```

ValidateCount Attribute Declaration

- The ValidateCount attribute specifies the minimum and maximum number of arguments allowed for a cmdlet parameter.

```
function Get-ValidateCount {
    Param(
        [Parameter(Mandatory)]
        [Validatecount(1,3)]
        [string[]]
        $ComputerName
    )
    Write-Output $ComputerName
}
Get-ValidateCount -ComputerName pc1,pc2,pc3
Get-ValidateCount -ComputerName pc1
Get-ValidateCount -ComputerName pc1,pc2,pc3,pc4
```

ValidateScript validation attribute

- The ValidateScript attribute specifies a script that is used to validate a parameter or variable value. PowerShell pipes the value to the script and generates an error if the script returns $false or if the script throws an exception.
- When you use the ValidateScript attribute, the value that's being validated is mapped to the $_ variable. You can use the $_ variable to refer to the value in the script.

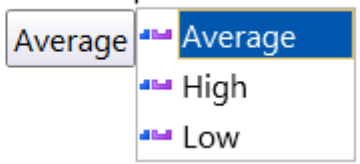
```
function Get-SodaPrice {
    param (
        [Parameter(Mandatory)]
        [ValidateScript({$_ -le 20})]
        $SodaPrice
    )
    Write-Output "Soda price is: $SodaPrice"
}
Get-SodaPrice -SodaPrice 10
Get-SodaPrice -SodaPrice 20
Get-SodaPrice -SodaPrice 30
```

ValidateSet validation attribute

- The ValidateSet attribute specifies a set of valid values for a parameter or variable and enables tab completion. PowerShell generates an error if a parameter or variable value doesn't match a value in the set

```
function Get-ValidateSet {
    Param(
        [Parameter(Mandatory)]
        [ValidateSet("Low", "Average", "High")]
        [string[]]
        $Detail
    )
    Write-Output $Detail
}
```

```
Get-ValidateSet -Detail |
```



Demonstration

Advanced attributes &
Validators

Questions?

Advanced attributes & Validators

[CmdletBinding()] Attribute

- The **CmdletBinding** attribute is an attribute of functions that makes them operate like compiled cmdlets written in C#. It provides access to the features of cmdlets.
- Parameter binding now checks for valid parameters
  - Will fail on unknown parameters
- Provides access to certain functionality within $PSCmdlet variable
- Automatically enables common parameters like -Verbose
- Optionally turn off positional binding
- **When using [CmdletBinding()] you MUST declare the param section.**

[CmdletBinding()] Attribute (continue)

- When using [CmdletBinding()] you have to declare the param section.
- You can use empty param section

```
Function Get-CmdletBinding
{
   [cmdletbinding(…)]
   Param ($parameter1,$parameterN)
}

Function Get-CmdletBinding
{
   [cmdletbinding(…)]
   Param ()
}
```

Demonstration

[CmdletBinding()] Attribute

[CmdletBinding()] Attribute - Risk Mitigation

```
function  <Name>
{
    [CmdletBinding(
        SupportsShouldProcess=<Boolean>,
        ConfirmImpact=<String>,
        DefaultParameterSetName=<String>,
        HelpURI=<URI>,
        SupportsPaging=<Boolean>,
        PositionalBinding=<Boolean>)]
    Param ()
}
```

Activates support for Risk Mitigation when set to $True

–Whatif and –Confirm common parameter now present

[CmdletBinding()] Attribute - Risk Mitigation cont.

```
function  <Name>
{
    [CmdletBinding(
        SupportsShouldProcess=<Boolean>,
        ConfirmImpact=<String>,
        DefaultParameterSetName=<String>,
        HelpURI=<URI>,
        SupportsPaging=<Boolean>,
        PositionalBinding=<Boolean>)]
    Param ()
}
```

- Sets general severity level of changes made by function
- Used in conjunction with $ConfirmPreference automatic variable to control confirmation behavior

[CmdletBinding()] - DefaultParameterSetName

```
function  <Name>
{
    [CmdletBinding(
        SupportsShouldProcess=<Boolean>,
        ConfirmImpact=<String>,
        DefaultParameterSetName=<String>,
        HelpURI=<URI>,
        SupportsPaging=<Boolean>,
        PositionalBinding=<Boolean>)]
    Param ()
}
```

Specifies the parameter set to use when it cannot be determined from user input

[CmdletBinding()] - HelpURI

```
function  <Name>
{
    [CmdletBinding(
        SupportsShouldProcess=<Boolean>,
        ConfirmImpact=<String>,
        DefaultParameterSetName=<String>,
        HelpURI=<URI>,
        SupportsPaging=<Boolean>,
        PositionalBinding=<Boolean>)]
    Param ()
}
```

- Specifies an online version of a help topic that describes the function
- Must begin with "http" or "https"

[CmdletBinding()] - SupportsPaging

```
function  <Name>
{
    [CmdletBinding(
        SupportsShouldProcess=<Boolean>,
        ConfirmImpact=<String>,
        DefaultParameterSetName=<String>,
        HelpURI=<URI>,
        SupportsPaging=<Boolean>,
        PositionalBinding=<Boolean>)]
    Param ()
}
```

- Automatically adds First, Skip, and IncludeTotalCount parameters
- Allow users to select output from a large result set
- Designed for functions that return data from data stores, such as a SQL database

[CmdletBinding()] - PositionalBinding

```
function  <Name>
{
    [CmdletBinding(
        SupportsShouldProcess=<Boolean>,
        ConfirmImpact=<String>,
        DefaultParameterSetName=<String>,
        HelpURI=<URI>,
        SupportsPaging=<Boolean>,
        PositionalBinding=<Boolean>)]
    Param ()
}
```

- Determines whether function parameters are positional by default
- Default value is $True when not present in CmdletBinding
- When parameters are positional, the parameter name is optional
- Position argument of the Parameter attribute takes precedence over the PositionalBinding argument

Demonstration

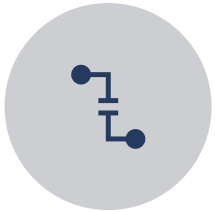
CmdeItBinding

Questions?

Streams

Understanding Streams

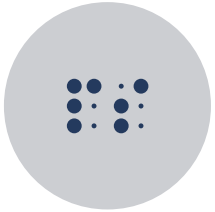
Streams separate different categories of output.



Without separation, error and output objects would be mixed, requiring later filtering.

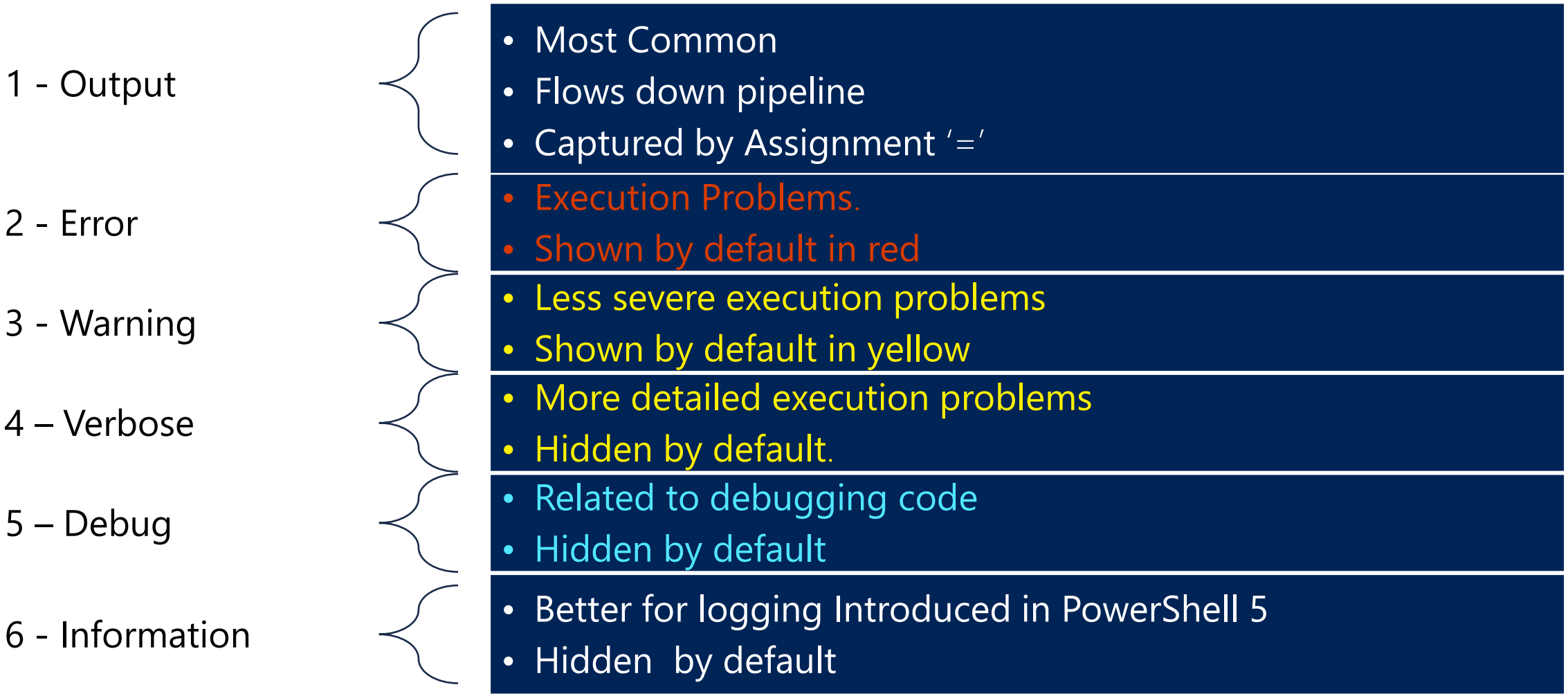
Some streams are hidden by default (Verbose, Debug, Information).

Redirection operators allow control over a stream's destination.



Write-* sends object to related streams.

The Streams


1 - Output
• Most Common
• Flows down pipeline
• Captured by Assignment '='
2 - Error
• Execution Problems.
• Shown by default in red
3 - Warning
• Less severe execution problems
• Shown by default in yellow
4 – Verbose
• More detailed execution problems
• Hidden by default.
5 – Debug
• Related to debugging code
• Hidden by default
6 - Information
• Better for logging Introduced in PowerShell 5
• Hidden by default

Stream Cmdlets

```
PS C:\> "Output Stream"
Output Stream

PS C:\> Write-Output "Output Stream"
Output Stream

PS C:\> Write-Error "Error Stream"
Write-Error "Error Stream" : Error Stream
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException

PS C:\> Write-Warning "Warning Stream"
WARNING: Warning Stream

PS C:\> Write-Verbose "Verbose Stream" -Verbose
VERBOSE: Verbose Stream

PS C:\> Write-Debug "Debug Stream" -Debug
DEBUG: Debug Stream

PS C:\> Write-Information "Information Stream" -InformationAction Continue
Information Stream
```

Controlling Stream Behavior

Preference Variables

- Variables that control how PowerShell reacts to stream messages

Cmdlet common parameters

- -Verbose, -Debug, -ErrorAction, -WarningAction

No preference variable for Output Stream

- Controlled by assignments, redirection, and pipeline

```
# Stream Defaults

PS C:\> $ErrorActionPreference
Continue

PS C:\> $WarningPreference
Continue

PS C:\> $VerbosePreference
SilentlyContinue

PS C:\> $DebugPreference
SilentlyContinue

PS C:\> $InformationPreference
SilentlyContinue
```

Redirecting Streams to Files (Output, Error, Warning)

Operator	Description
>	Sends **output** to specified file
>>	Appends **output** to contents of specified file
2>	Sends **errors** to specified file
2>>	Appends **errors** to contents of specified file
2>&1	Sends **errors** (2) and success **output** (1) to success **output** stream
3>	Sends **warnings** to specified file
3>>	Appends **warnings** to contents of specified file
3>&1	Sends **warnings** (3) and success **output** (1) to success **output** stream

Redirecting Streams to Files (Verbose, Debug)

Operator	Description
4>	Sends **verbose** output to specified file
4>>	Appends **verbose** output to contents of specified file
4>&1	Sends **verbose** output (4) and success output (1) to success output stream
5>	Sends **debug** messages to specified file
5>>	Appends **debug** messages to contents of specified file
5>&1	Sends **debug** messages (5) and success **output** (1) to success output stream

Redirect everything

Operator	Description
*>	Sends all output types to specified file
*>>	Appends all output types to contents of specified file
*>&1	Sends all output types (*) to success output stream

Demonstration

Streams

Questions?