Outline

In this lab, we'll discuss `BEGIN`, `PROCESS`, and `END` blocks.

Exercise 1 - Create a Process Block

Processing Pipeline Input

The process block allows us to process input from the pipeline and loop through each item if multiple items are received. Let's create a `PROCESS` block for Initialize-OutputPath.ps1.

```
Function Initialize-OutputPath
{
    # .EXTERNALHELP
    [CmdletBinding(SupportsShouldProcess = $true, ConfirmImpact = "High",
DefaultParameterSetName = "Folder")]
    PARAM
    (
        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "Folder",
ValueFromPipeline = $true)]
        [ValidateNotNullOrEmpty()]
        [Alias("OutputPath")]
        $FolderPath,

        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "File")]
        [ValidateNotNullOrEmpty()]
        $FilePath
    )

    PROCESS
    {
        $NewPath = SWITCH ($PSCmdlet.ParameterSetName)
        {
            "Folder" {$FolderPath}
            "File"   {$FilePath}
        }
        $ItemType = SWITCH ($PSCmdlet.ParameterSetName)
        {
            "Folder" {"Directory"}
            "File"   {"File"}
        }
        IF ($PSCmdlet.ShouldProcess($NewPath, "Create
$($PSCmdlet.ParameterSetName)"))
        {
            Write-Verbose "Testing $NewPath..."
```

```
            SWITCH (Test-Path $NewPath)
            {
                $false
                {
                    Write-Verbose "$($PSCmdlet.ParameterSetName) $NewPath does not
exist. Creating...."
                    New-Item -ItemType $ItemType -Path $NewPath -
WhatIf:$WhatIfPreference -Verbose:$Global:VerbosePreference
                    Write-Verbose "Success!"
                }
                $true
                {
                    Write-Warning "$($PSCmdlet.ParameterSetName) $NewPath already
exists."
                    Get-Item -Path $NewPath -Verbose:$Global:VerbosePreference
                }
            }
        }
    }
}
```

Now after we've updated our module, we should be able to process input from the pipeline, right? Let's try it out.

```
# Before running this, make sure your PowerShell working directory is the same
folder as the file you're currently reading!
"MyFolder" | Initialize-OutputPath -Confirm:$false
```

The function didn't accept the data from the pipeline, and still prompts for input. This is because we haven't allowed any parameter to accept input from the pipeline yet. Let's assign pipeline input to our `FolderPath` parameter by adding the `ValueFromPipeline` argument to the parameter attribute.

```
[Parameter(Mandatory = $true, Position = 0, ParameterSetName = "Folder",
ValueFromPipeline = $true)]
[ValidateNotNullOrEmpty()]
[Alias("OutputPath")]
$FolderPath,
```

Now our function should accept pipeline input, including for multiple items.

```
# Before running this, make sure your PowerShell working directory is the same
folder as the file you're currently reading!
"MyFolder","MyFolder\1","MyFolder\2" | Initialize-OutputPath -Confirm:$false
```

We can also accept pipeline input by property names by using the `ValueFromPipelineByPropertyName` argument. Let's add that to our `FilePath` parameter. We'll also have to remove the pipeline input from our `FolderPath` parameter since it's a member of a different ParameterSet, and the function won't know what to default to. If we had multiple parameters in the same ParameterSet, we could also assign each of them to accept a value by property name.

```
[Parameter(Mandatory = $true, Position = 0, ParameterSetName = "Folder")]
[ValidateNotNullOrEmpty()]
[Alias("OutputPath")]
$FolderPath,

[Parameter(Mandatory = $true, Position = 0, ParameterSetName = "File",
ValueFromPipelineByPropertyName = $true)]
[ValidateNotNullOrEmpty()]
$FilePath
```

Now let's create an object that has a property called `FilePath` to pass to our function and try to create some files.

```
# Before running this, make sure your PowerShell working directory is the same
folder as the file you're currently reading!
# Declare variables
$NewFiles = @()
$FilePaths = "\1\1.txt", "\1\2.txt", "\2\1.txt", "\2\2.txt"

# Create custom object
$FilePaths | ForEach-Object {
    $NewFiles += [PSCustomObject]@{
        FilePath     = "MyFolder" + $PSItem
        JunkProperty = "FakeData"
    }
}

# Create Files
$NewFiles | Initialize-OutputPath -Confirm:$false -Verbose
```

Success! This approach is extremely useful if you want the ability to be able to pass an object from the pipeline, but only process certain properties of that object as an argument; our `JunkProperty` property was ignored entirely by the function. We also overrode our default ParameterSetName by using the pipeline; if we ran `Initialize-OutputPath MyFolder`, the function would still default to creating a folder.

```
Initialize-OutputPath MyFolder -WhatIf
```

It's important to note that when you allow pipeline input by property, you can't pass raw data like strings from the pipeline: it always expects a property and will throw a terminating error if one isn't found. We can call the property of `$NewFiles` object directly and try to pass it to `Initialize-OutputPath` to show this behavior.

```
$NewFiles.FilePath | Initialize-OutputPath -WhatIf
```

It's also important to note that `ValueFromPipeline` and `ValueFromPipelineByPropertyName` are not compatible on the same line at the time this is written.

Processing Multiple Items

What if we wanted to create multiple folders without using the pipeline input? Let's try it out:

```
Initialize-OutputPath MyFolder,MyFolder2 -Confirm:$false
```

The function only created one folder because the `PROCESS` block only iterates over **pipeline** input. We'll need to create a `FOREACH` loop inside our function, and then `strongly type` our parameters as arrays of strings to allow multiple items to be processed. PowerShell's a very forgiving language, and will implicitly assign a type based on what is detected, but in our case we want to explicitly declare the type. `[String]` before the parameter name would declare a single object of type string, and [String[]] before it will allow for an array of strings to be passed.

Let's update our function with those changes:

```
Function Initialize-OutputPath
{
    # .EXTERNALHELP
    [CmdletBinding(SupportsShouldProcess = $true, ConfirmImpact = "High",
DefaultParameterSetName = "Folder")]
    PARAM
    (
        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "Folder",
ValueFromPipeline = $true)]
        [ValidateNotNullOrEmpty()]
        [Alias("OutputPath")]
        [String[]]$FolderPath,

        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "File")]
        [ValidateNotNullOrEmpty()]
        [String[]]$FilePath
    )

    PROCESS
    {
```

```
        $NewPaths = SWITCH ($PSCmdlet.ParameterSetName)
        {
            "Folder" {$FolderPath}
            "File"   {$FilePath}
        }
        $ItemType = SWITCH ($PSCmdlet.ParameterSetName)
        {
            "Folder" {"Directory"}
            "File"   {"File"}
        }
        FOREACH ($NewPath in $NewPaths)
        {
            IF ($PSCmdlet.ShouldProcess($NewPath, "Create
$($PSCmdlet.ParameterSetName)"))
            {
                Write-Verbose "Testing $NewPath..."
                SWITCH (Test-Path $NewPath)
                {
                    $false
                    {
                        Write-Verbose "$($PSCmdlet.ParameterSetName) $NewPath does
not exist. Creating...."
                        New-Item -ItemType $ItemType -Path $NewPath -
WhatIf:$WhatIfPreference -Verbose:$Global:VerbosePreference
                        Write-Verbose "Success!"
                    }
                    $true
                    {
                        Write-Warning "$($PSCmdlet.ParameterSetName) $NewPath
already exists."
                        Get-Item -Path $NewPath -Verbose:$Global:VerbosePreference
                    }
                }
            }
        }
    }
}
```

We also changed our pipeline input back to folders. Now we can create folders by running either of the following:

```
"MyFolder","MyFolder2" | Initialize-OutputPath -Confirm:$false
Initialize-OutputPath "MyFolder","MyFolder2" -Confirm:$false
```

Exercise 2 - Create a Begin Block

A `BEGIN` block is an optional complement to the `PROCESS` block; it's useful for one-time processing operations in the event that multiple objects are received from the pipeline. As the name implies, it's placed before the `PROCESS` block. Let's add one to our function, and then move our `$ItemType` declaration into it.

```
Function Initialize-OutputPath
{
    # .EXTERNALHELP
    [CmdletBinding(SupportsShouldProcess = $true, ConfirmImpact = "High",
DefaultParameterSetName = "Folder")]
    PARAM
    (
        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "Folder",
ValueFromPipeline = $true)]
        [ValidateNotNullOrEmpty()]
        [Alias("OutputPath")]
        [String[]]$FolderPath,

        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "File")]
        [ValidateNotNullOrEmpty()]
        [String[]]$FilePath
    )

    BEGIN
    {
        $ItemType = SWITCH ($PSCmdlet.ParameterSetName)
        {
            "Folder" {"Directory"}
            "File"   {"File"}
        }
        Write-Verbose "ParameterSetName is $Itemtype."
    }
    PROCESS
    {
        $NewPaths = SWITCH ($PSCmdlet.ParameterSetName)
        {
            "Folder" {$FolderPath}
            "File"   {$FilePath}
        }
        FOREACH ($NewPath in $NewPaths)
        {
            IF ($PSCmdlet.ShouldProcess($NewPath, "Create
$($PSCmdlet.ParameterSetName)"))
            {
                Write-Verbose "Testing $NewPath..."
                SWITCH (Test-Path $NewPath)
                {
                    $false
                    {
                        Write-Verbose "$($PSCmdlet.ParameterSetName) $NewPath does
```

```
not exist. Creating...."
                        New-Item -ItemType $ItemType -Path $NewPath -
WhatIf:$WhatIfPreference -Verbose:$Global:VerbosePreference
                        Write-Verbose "Success!"
                    }
                    $true
                    {
                        Write-Warning "$($PSCmdlet.ParameterSetName) $NewPath
already exists."
                        Get-Item -Path $NewPath -Verbose:$Global:VerbosePreference
                    }
                }
            }
        }
    }
}
```

Now when we run our function with verbose messaging, we'll only receive one message stating the item type, but all the folders will be processed.

Exercise 3 - Create an End Block

Just like the `BEGIN` block, the `END` block handles one-time processing of operations, but instead executes after the `PROCESS` block.

Let's create a `$Results` object that we'll add to each time our `FOREACH` loop runs, and then output our results in the `END` block.

```
Function Initialize-OutputPath
{
    # .EXTERNALHELP
    [CmdletBinding(SupportsShouldProcess = $true, ConfirmImpact = "High",
DefaultParameterSetName = "Folder")]
    PARAM
    (
        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "Folder",
ValueFromPipeline = $true)]
        [ValidateNotNullOrEmpty()]
        [Alias("OutputPath")]
        [String[]]$FolderPath,

        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "File")]
        [ValidateNotNullOrEmpty()]
        [String[]]$FilePath
    )
```

```
    BEGIN
    {
        $Results = [System.Collections.ArrayList]::new()
        $ItemType = SWITCH ($PSCmdlet.ParameterSetName)
        {
            "Folder" {"Directory"}
            "File"   {"File"}
        }
        Write-Verbose "ParameterSetName is $Itemtype."
    }
    PROCESS
    {
        $NewPaths = SWITCH ($PSCmdlet.ParameterSetName)
        {
            "Folder" {$FolderPath}
            "File"   {$FilePath}
        }
        FOREACH ($NewPath in $NewPaths)
        {
            IF ($PSCmdlet.ShouldProcess($NewPath, "Create
$($PSCmdlet.ParameterSetName)"))
            {
                Write-Verbose "Testing $NewPath..."
                $NewItem = SWITCH (Test-Path $NewPath)
                {
                    $false
                    {
                        Write-Verbose "$($PSCmdlet.ParameterSetName) $NewPath does
not exist. Creating...."
                        New-Item -ItemType $ItemType -Path $NewPath -
WhatIf:$WhatIfPreference -Verbose:$Global:VerbosePreference
                        Write-Verbose "Success!"
                    }
                    $true
                    {
                        Write-Warning "$($PSCmdlet.ParameterSetName) $NewPath
already exists."
                        Get-Item -Path $NewPath -Verbose:$Global:VerbosePreference
                    }
                }
                [void]$Results.Add($NewItem)
            }
        }
    }
    END
    {
        Return $Results
    }
}
```

Now let's try processing some folders.

```
"MyFolder","MyFolder2" | Initialize-OutputPath -Confirm:$false -Verbose
```

We can see that our function output all the success streams at the end of execution instead of at the end of each loop in the `PROCESS` block. The one drawback of the `END` block is that if the function fails due to a terminating error, the `END` block never executes. Because of this, it can sometimes be a better approach to output your results at the end of the `PROCESS` block.

Exercise 4 - Create a Clean Block

In PowerShell 7.3, Microsoft added a `CLEAN` block that can be used to help perform clean-up operations, regardless of whether the function encounters a terminating error. It can be used to close connections that would otherwise be left open, remove temporary files, or output additional messaging. However, it's important to note that any data sent to the success stream is discarded by the `CLEAN` block, so you'll be unable to use it to output results like in our function above.

Let's set an `ErrorActionPreference` value of `Stop` to inside our function so that all errors are treated as terminating, add a clean block, and add a Requires statement.

```
#requires -Version 7.3
Function Initialize-OutputPath
{
    # .EXTERNALHELP
    [CmdletBinding(SupportsShouldProcess = $true, ConfirmImpact = "High",
DefaultParameterSetName = "Folder")]
    PARAM
    (
        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "Folder",
ValueFromPipeline = $true)]
        [ValidateNotNullOrEmpty()]
        [Alias("OutputPath")]
        [String[]]$FolderPath,

        [Parameter(Mandatory = $true, Position = 0, ParameterSetName = "File")]
        [ValidateNotNullOrEmpty()]
        [String[]]$FilePath
    )

    BEGIN
    {
        $ErrorActionPreference = 'Stop'
        $Results = [System.Collections.ArrayList]::new()
        $ItemType = SWITCH ($PSCmdlet.ParameterSetName)
```

```
        {
            "Folder" {"Directory"}
            "File"   {"File"}
        }
        Write-Verbose "ParameterSetName is $Itemtype."
    }
    PROCESS
    {
        $NewPaths = SWITCH ($PSCmdlet.ParameterSetName)
        {
            "Folder" {$FolderPath}
            "File"   {$FilePath}
        }
        FOREACH ($NewPath in $NewPaths)
        {
            IF ($PSCmdlet.ShouldProcess($NewPath, "Create
$($PSCmdlet.ParameterSetName)"))
            {
                Write-Verbose "Testing $NewPath..."
                $NewItem = SWITCH (Test-Path $NewPath)
                {
                    $false
                    {
                        TRY
                        {
                            Write-Verbose "$($PSCmdlet.ParameterSetName) $NewPath
does not exist. Creating...."
                            New-Item -ItemType $ItemType -Path $NewPath -
WhatIf:$WhatIfPreference -Verbose:$Global:VerbosePreference
                            Write-Verbose "Success!"
                        }
                        CATCH
                        {
                            Throw
                        }
                    }
                    $true
                    {
                        Write-Warning "$($PSCmdlet.ParameterSetName) $NewPath
already exists."
                        Get-Item -Path $NewPath -Verbose:$Global:VerbosePreference
                    }
                }
                [void]$Results.Add($NewItem)
            }
        }
    }
    END
    {
        Write-Verbose "All executions were successful."
```

```
        Return $Results
    }
    CLEAN
    {
        Write-Verbose "The clean block still executes regardless of whether the
function succeeds."
        Write-Verbose "$($Results | Out-String)"
    }
}
```

Now we'll use some illegal characters and force the function to fail.

```
"MyFolder","???","MyFolder2" | Initialize-OutputPath -Confirm:$false -Verbose
```

We can see from our verbose messaging that the clean block still executed, even though `MyFolder2` was never processed and the `END` block never executed.

You can read more about the `BEGIN`, `PROCESS`, `END`, and `CLEAN` blocks here.