Outline

In this lab, we'll discuss CmdletBinding and how to use it.

Exercise 1 - Create a Function with CmdletBinding

Let's start by modifying our `Initialize-OutputPath` function. Open Initialize-OutputPath.ps1 and replace the contents with the text below.

```
Function Initialize-OutputPath
{
    [CmdletBinding(SupportsShouldProcess = $true, ConfirmImpact = "High")]
    PARAM
    (
        [Parameter(Mandatory = $true)]
        [ValidateNotNullOrEmpty()]
        [Alias("OutputPath")]
        $FolderPath
    )

    IF ($PSCmdlet.ShouldProcess($FolderPath))
    {
        IF (!(Test-Path $FolderPath))
        {
            New-Item -ItemType Directory -Path $FolderPath
        }
    }
}
```

If we compare our new `Initialize-Module` function to the old one, we'll note a few differences:

- We've added a line containing the `CmdletBinding` attribute along with a few arguments: `SupportsShouldProcess`and `ConfirmImpact`.
- We've added another `IF` statement containing `$PSCmdlet.ShouldProcess`.
- We've changed our `OutputPath` parameter's name to `FolderPath`, given it an alias, added some validation, and made the parameter mandatory. The `Alias` attribute is extremely useful because it allows you to use **either** name for your parameter; this is handy when making changes because it allows for backward compatibility.

So what does `CmdletBinding` do? In short, it treats your function more like a compiled cmdlet, and this provides us a ton of benefits not otherwise available, like confirmation prompts, alternative output streams, positional parameter binding, and more. You can see the feature list here.

Exercise 2 - Prompt for Confirmation

Go ahead and import the module, then run `Initialize-OutputPath` without any arguments.

```
# Before running this, make sure your PowerShell working directory is the same
folder as the file you're currently reading!
$ModuleName = "06_Split-Shakespeare"
Import-Module ".\$ModuleName\$ModuleName.psd1" -Force

Initialize-OutputPath
```

Since we made our parameter mandatory, the function will prompt for a file path: let's supply a value of `MyFolder`. Now we'll be greeted by some new behavior: we received a confirmation message asking if we want to proceed. In this case, let's confirm with `Y`, or by hitting `Enter`. Now we've created a new folder named `MyFolder`. You can delete this folder now either by using the GUI or the `Remove-Item` cmdlet.

So why did we receive a confirmation prompt? Three reasons:

- We enabled `SupportsShouldProcess` as an argument in our `CmdletBinding`.
- We set a confirmation preference of `High`, which is equal to the default `ConfirmPreference` value. `ConfirmPreference` is an Automatic Variable, and we can confirm (or set) it by invoking `$ConfirmPreference`. If we lower our confirmation preference to anything lower than high, we'll start receiving more confirmation messages from various cmdlets; this can cause unintended behavior in scripts, so it's generally a safer practice to set the `ConfirmImpact` to `High` in functions. You can read more about `ConfirmPreference` here.
- We nested our `New-Item` command inside an `IF` statement that evaluated for `$PSCmdlet.ShouldProcess`. Anything nested in this `IF` statement will not execute unless confirmation is given, or the `ConfirmImpact` value of the function is set to a lower value than `$ConfirmPreference`.

Let's customize our confirmation prompt a bit. Change the `IF` statement to the following:

```
IF ($PSCmdlet.ShouldProcess($FolderPath, "Create Folder"))
```

Save the function and force re-import the module. Let's again run `Initialize-OutputPath`, supplying the same folder name. This time, the confirmation will list the operation as `Create Folder` instead of the name of the function itself. This is because we supplied an operational message as the second argument to our `ShouldProcess` method. As an additional nicety, these targets and operations also work with the `WhatIf` parameter. You can rerun the function, appending the `WhatIf` parameter, and you'll receive the same message, but the function won't take any action.

If you wanted to take things a step further and create a completely custom message, you can add a third argument. Modify the `IF` statement, save, force re-import, and run as we did before.

```
IF ($PSCmdlet.ShouldProcess($FolderPath, "Create Folder", "This will create a new
folder named $FolderPath. Are you sure?"))
```

As you can see on the latest run, we have a custom confirmation message, but you should note that it doesn't also apply to the `WhatIf` switch.

If we didn't want to receive the confirmation message when invoking the command, but we wanted the default behavior to prompt, all we have to do is run our `Initialize-OutputPath` function with the `-Confirm:$false` parameter. You could also implement a switch called `-Force` and evaluate against that, but that falls outside the scope of this exercise. You can read about it and more here.

Exercise 3 - Use Alternate Output Streams

Before we explore what output streams do, we can make some small quality of life improvements. `CmdletBinding` also allows us to use positional binding for parameters, meaning we don't have to name the parameter, but instead can assign an ordered "slot" for our arguments. That means instead of typing `Initialize-OutputPath -FolderPath MyFolder`, we can just type `Initialize-OutputPath MyFolder` and the function will handle the rest. To use it, all we have to do is assign a position as an argument to our `Parameter` attribute, starting from zero.

In practice, our parameter attribute will look like this:

```
[Parameter(Mandatory = $true, Position = 0)]
```

Update the function and try it out. Now... On to output streams.

Warning Streams

Let's replace our `IF` statement that contains the `Test-Path` cmdlet and replace it with the following `SWITCH` statement:

```
SWITCH (Test-Path $FolderPath)
{
    $false {New-Item -ItemType Directory -Path $FolderPath -
WhatIf:$WhatIfPreference}
    $true  {Write-Warning "Folder $FolderPath already exists."}
}
```

After we've updated our module, let's run the function. If the `MyFolder` folder is left over from previous runs, you'll receive a warning message. This warning isn't part of the default output, but is instead it's own output

stream. We can easily illustrate this by capturing the output as a variable.

```
$Folder = Initialize-OutputPath MyFolder -Confirm:$false
$Folder
```

Since the folder already exists, we didn't actually perform any action that would've been reported to the default `Success` stream; we only issued a message to the `Warning` stream. Because of this, when we invoke our `$Folder` variable, we don't receive any output in return.

If we wanted to capture output regardless of whether the folder exists, we only have to add a line inside our `SWITCH` statement to get the existing folder. The brackets have been adjusted in the statement for readability.

```
SWITCH (Test-Path $FolderPath)
{
    $false
    {
        New-Item -ItemType Directory -Path $FolderPath -WhatIf:$WhatIfPreference
    }
    $true
    {
        Write-Warning "Folder $FolderPath already exists."
        Get-Item -Path $FolderPath
    }
}
```

Now when we run the function, we'll receive the same default output regardless of whether the folder exists, while retaining our warning message being output to the screen.

Verbose Stream

If we wanted to add more information as feedback, we can use the `Verbose` stream to do so. Let's add some verbose messaging to our function:

```
Function Initialize-OutputPath
{
    [CmdletBinding(SupportsShouldProcess = $true, ConfirmImpact = "High")]
    PARAM
    (
        [Parameter(Mandatory = $true, Position = 0)]
        [ValidateNotNullOrEmpty()]
        [Alias("OutputPath")]
        $FolderPath
    )
```

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

Now if we run our function with the `-Verbose` switch, we'll receive even more messaging. Delete the MyFolder folder, then try running the function twice to see the difference.

```
$NewFolder = Initialize-OutputPath MyFolder -Confirm:$false -Verbose
$ExistingFolder = Initialize-OutputPath MyFolder -Confirm:$false -Verbose

Compare-Object $NewFolder $ExistingFolder -IncludeEqual
```

As we can see, the output of both our variables is identical, but we've received different verbose/warning messaging depending on the actions taken. We've also suppressed the output from the `New-Item` and `Get-Item` cmdlets by adding `-Verbose:$Global:VerbosePreference` to the end of each command. The reason this was done is because **all** commands inside the function inherit the `VerbosePreference` for the scope. What we've done is manually override the preference for those lines with the value of our `VerbosePreference` from the `Global` scope, which is at the root of our PowerShell session. This also means you could toggle your `VerbosePreference` from the default value of `'SilentlyContinue'` to `'Continue'` and receive even more messaging from the commands themselves, if they support verbose messaging. Just remember to set it back when you're done, or you'll have quite a lot of feedback in your session!

Delete the MyFolder folder and try it out.

```
$VerbosePreference = 'Continue'
Initialize-OutputPath MyFolder -Confirm:$false -Verbose
```

```
$VerbosePreference = 'SilentlyContinue'
Initialize-OutputPath MyFolder -Confirm:$false -Verbose
```

Using verbose messaging can be extremely useful when troubleshooting scripts or functions. You could also use the `Debug` stream to prompt for continuance, but that falls outside the scope of this exercise. There are actually 7 output streams in all, 6 of which are numbered. You can read more about them here.

Exercise 4 - Use ParameterSets

So far our `Initialize-OutputPath` function is fairly useful, but what if we wanted to use it for files as well? We can use `ParameterSets` to accomplish that with ease. Before we attempt to integrate this into our `Initialize-OutputPath` function, let's create a simple function to explore the behavior. Copy and paste this function into your PowerShell session.

```
Function Test-ParameterSets
{
    [CmdletBinding()]
    PARAM
    (
        [Parameter(ParameterSetName = "Apples")]
        [Switch]$ShowApples,

        [Parameter(ParameterSetName = "Oranges")]
        [Switch]$ShowOranges
    )

    Return $PSCmdlet.ParameterSetName
}
```

When we declared each parameter, we assigned each a `ParameterSetName`. ParameterSets are mutually exclusive, meaning they can't be combined; the function will terminate with an error before it ever attempts to execute the contents if you try. Let's do so, just to see.

```
Test-ParameterSets -ShowApples -ShowOranges
```

If we run one or the other, the function will simply output the name of the ParameterSet we've assigned the parameter to.

```
Test-ParameterSets -ShowApples
Test-ParameterSets -ShowOranges
```

You can create as many ParameterSets as you want. You can also have as many parameters as you want assigned to a ParameterSet, each with their own attributes (Mandatory, Position, Validation, etc.), but a parameter can only be assigned to **one** ParameterSet. If you wanted to share a parameter between multiple ParameterSets, there's a way to do so that doesn't *technically* break this rule.

First, let's try running our function without specifying any parameters.

```
Test-ParameterSets
```

Now we've received an error stating that PowerShell can't determine the ParameterSet to use. This is expected behavior; we haven't assigned a default. We can supply this information as an argument inside our `CmdletBinding` attribute. Modify the function with the following line and paste it into your PowerShell session:

```
[CmdletBinding(DefaultParameterSetName = "Apples")]
```

Now when we rerun our function, we can see it defaults to `Apples`. This doesn't necessarily let us split parameters between ParameterSets, but it lays the foundation. Inside PowerShell, there's a reserved ParameterSetName called `__AllParameterSets`, and we can set it as our default. Modify the function and run it again.

```
[CmdletBinding(DefaultParameterSetName = "__AllParameterSets")]
```

Now we get a valid return, even though we didn't assign a parameter to that ParameterSetName. Microsoft's documentation is actually wrong when they state this has no effect, as you can see from our example. But what does this reserved ParameterSet do, and why does it matter? It allows us to create parameters without specifying a ParameterSetName, and they are shared by **all** ParameterSets.

From a technical standpoint, the parameter will be assigned to the `__AllParameterSets` ParameterSet, which is shared by all. You could also explicitly assign the parameter if you desired, and it would have the same effect. Let's see it in practice.

```
Function Test-ParameterSets
{
    [CmdletBinding(DefaultParameterSetName = "__AllParameterSets")]
    PARAM
    (
        [Parameter(ParameterSetName = "Apples")]
        [Switch]$ShowApples,

        [Parameter(ParameterSetName = "Oranges")]
        [Switch]$ShowOranges,
```

```
        [Parameter()]
        [Switch]$ShowBananas
    )

    Return $PSCmdlet.ParameterSetName
}
```

We've created a third parameter called `ShowBananas` that doesn't have an assignment, so it should be shared by all the ParameterSets.

```
Test-ParameterSets -ShowApples -ShowBananas
Test-ParameterSets -ShowOranges -ShowBananas
Test-ParameterSets -ShowBananas
Test-ParameterSets
```

Now that we've explored the concept, we can apply it to our `Initialize-OutputPath` function. This time, we'll do the following:

- Add a `FilePath` parameter
- Assign each parameter it's own ParameterSetName
- Set the default ParameterSetName to `Folder`
- Set new variables based on the ParameterSetName
- Modify the workflow to use the new variables

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

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

    $NewPath = SWITCH ($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
            }
        }
    }
}
```

Delete MyFolder and try to create both a folder and a file to confirm the behavior we expect.

```
Initialize-OutputPath MyFolder -Confirm:$false -Verbose
Initialize-OutputPath -FilePath MyFolder\MyFile.txt -Confirm:$false -Verbose
```

Now that we've confirmed everything works as expected, update the module manifest to version 1.2.0 to reflect our improvements.

You can read more about ParameterSets here.