Outline

In this lab, we'll discuss how to author, consume, and export help for PowerShell functions.

Exercise 1 - Create Comment-Based Help

Much like with the pre-compiled PowerShell cmdlets, functions and modules also support `Get-Help` and generating documentation for your tools is quick and easy: all we have to do is place our documentation inside a comment block at the beginning of our function. There are a few keywords that we'll need to add to make things work, but the basic syntax is below in a sample function named `Test-MyHelp`.

```
Function Test-MyHelp
{
    <#
        .SYNOPSIS
        A brief description of what the function does.

        .DESCRIPTION
        A longer description of what the function does. Go into as much detail as
you believe is necessary.

        .PARAMETER MyParameterName
        A description of what the parameter is or does. The object type and
attributes will be automatically generated for you, so they don't need to be
included here. You'll need to create a .PARAMETER section for each of your
parameters.

        .INPUTS
        The .NET classes supported as input from the pipeline, if any.

        .OUTPUTS
        The .NET classes returned as output by your function, if any.

        .EXAMPLE
        Test-MyHelp

        A description of what your example does. You can create as many examples as
you want.

        .EXAMPLE
        Test-MyHelp -MyParameterName "Hello World!"

        Another example section... As an example.

        .NOTES
        Any additional information you'd like to include.

        .LINK
        A URL to your project repository or any other source of documentation, if
applicable.
    #>
    PARAM
    (
        [CmdletBinding()]
        [Parameter(Mandatory = $true)]
        [String]$MyParameterName
    )
}
```

While the function itself doesn't do anything, it's enough to interact with the `Get-Help` cmdlet in a live PowerShell session. Let's paste it into a PowerShell window and try it out.

```
Get-Help -Full Test-MyHelp
```

We can see our help was properly generated with all the information we entered, along with automatically generated syntax, remarks, and attributes for our parameter. And because we've included CmdletBinding, help was also generated indicating that common parameters like `-Verbose` are supported. The most common fields were chosen for this example, but there are more you can read about here.

It's really that simple. Now all we have to do is create help for each of our functions, but for this exercise we'll just focus on `Initialize-OutputPath`.

Open Initialize-OutputPath.ps1 and add the following help to the function, then update your module; this time we'll change the version to 1.2.1 since the change we made was so minor.

```
<#
    .SYNOPSIS
    Creates files and folders.

    .DESCRIPTION
    Creates files or folders for a given path if they do not already exist. If the
path already exists, a warning will be issued, and the existing object will be
returned as output.

    .PARAMETER FolderPath
    The path to the folder to create. Relative and absolute paths are both
supported.

    .PARAMETER FilePath
    The path to the file to create. Relative and absolute paths are both supported.
    .INPUTS
    None.

    .OUTPUTS
    System.IO.DirectoryInfo

    .EXAMPLE
    Initialize-OutputPath -FolderPath MyFolder -Confirm:$false

    Creates a folder named 'MyFolder' in your current working directory if it
doesn't already exist, without prompting for confirmation

    .EXAMPLE
    Initialize-OutputPath -FilePath C:\Users\Administrator\Documents\MyFile.txt

    Creates a file named 'MyFile.txt' at the absolute path
'C:\Users\Administrator\Documents' if it doesn't already exist
#>
```

Import the module and try getting help for `Initialize-OutputPath`.

Exercise 2 - Generate External Help

Comment-based help is a fantastic way of creating self-contained documentation for functions, but what if you wanted to store it externally as a separate file? Microsoft not only supports this, but offers tools to export existing comment-based help to their MAML format. There's an open-source tool called platyPS that allows you to generate both Markdown and MAML-formatted XML documents based on existing functions or modules.

First, we'll need to install it.

```
Install-Module platyPS
```

Now we need to generate our documentation. It's a two-step process: first we generate the Markdown file from the function or module, then generate the XML file from the Markdown file. We'll need to create a localized-language folder (in our case "en-US") in our module directory, then place our files in it. On import, the module will automatically search for an XML file with the same name as the module manifest with `-help` appended. So in our case the file will be named `07_Split-Shakespeare-help.xml`

```
# Before running this, make sure your PowerShell working directory is the same
folder as the file you're currently reading!

# Import existing module
```

```
Import-Module ".\07_Split-Shakespeare\07_Split-Shakespeare.psd1" -Force

# Generate documentation
$OutputFolder = "07_Split-Shakespeare\en-US"
$MarkdownHelp = New-MarkdownHelp -Command Initialize-OutputPath -OutputFolder
$OutputFolder
$XMLHelp = New-ExternalHelp -Path $MarkdownHelp -OutputPath "$OutputFolder\07_Split-
Shakespeare-help.xml"

# Housekeeping
Remove-Item $MarkdownHelp
```

Now we should have a valid help file, and we can delete the comment-based help from inside our `Initialize-OutputPath` function. Instead, we'll replace it with a single line:

```
# .EXTERNALHELP
```

If we had populated all our functions with comment-based help, we'd have used the `-Module` parameter instead of `-Command` in our `New-MarkdownHelp` command, and then done the same cleanup actions to remove our old comment-based help from each function.

Now if we force re-import our module, we should see the same help as before, except it's now being generated from our XML file.

```
Import-Module "07_Split-Shakespeare\07_Split-Shakespeare.psd1"
Get-Help -Full Initialize-OutputPath
```

This also opens up the possibility of making our help online-updatable by using the `HelpInfoURI` field in our module manifest. If you're interested in doing so, there's a fantastic blog entry on how it works here.

You can find more info on how to use PlatyPS here and here.