Outline

In this lab, we'll discuss monolithic vs non-monolithic modules, public vs. private functions, and module manifests.

Exercise 1 - Create a Module Manifest

If you open 05_Split-Shakespeare.psm1, you'll see that the helper functions from our `Split-Shakespeare` have been moved outside the function itself, which changes their scope from the `Function` scope to the `Script` scope. As we discussed in the last exercise, when we import a module, functions from the `Script` scope then get transferred to our `Local` scope. This means that the helper functions that previously only existed inside our `Split-Shakespeare` function are now exposed to our PowerShell session. Let's import our module and explore this behavior.

```
# You may have to modify the path below to reflect your local directory
Import-Module .\05_Split-Shakespeare.psm1
```

We can verify our module has been imported by using the `Get-Module` cmdlet, and then view the exposed commands by using the `Get-Command` cmdlet.

```
Get-Module -Name 05_Split-Shakespeare
Get-Command -Module 05_Split-Shakespeare
```

Now we have our helper functions imported, and we can call them in any script or function we want to develop. But what if we only wanted to expose a small subset of the functions in our module? We can accomplish this and more by using a `Module Manifest`. A manifest is a descriptor file that contains metadata such as the author, description, the path to module file, versioning, etc., and creating one is easy with the `New-ModuleManifest` cmdlet. In the example below, we'll supply a lot of information to the cmdlet via parameters, so instead of keeping it all on one line we'll use a technique called `Splatting` to pass a hash table to the cmdlet.

```
# You may have to modify any paths below to reflect your local directory
$Parameters =
@{
    Path          = ".\05_Split-Shakespeare.psd1"
    RootModule    = ".\05_Split-Shakespeare.psm1"
    Author        = "PowerShell Summit 2025"
    Description   = "Splits the complete works of Shakespeare into different files."
    ModuleVersion = "1.0.0"

}
```

```
New-ModuleManifest @Parameters
```

Splatting is a simple concept; we create a hashtable with the keys being the names of the parameters we want to use, and store it as a variable, then pass the hashtable to our cmdlet using the at symbol `@` instead of recalling the variable like we normally would with a dollar sign `$`. You can read more about splatting here.

Now we should have a new `.psd1` file with the same name as our .psm1 file in the same directory. If you open the manifest file, you'll see all the information we supplied has been pre-populated, along with several other new fields. If we only wanted to make certain functions visible to other functions or scripts outside the module, we can edit the `FunctionsToExport` field either manually or by using the `Update-ModuleManifest` cmdlet.

```
# You may have to modify the path below to reflect your local directory
Update-ModuleManifest -Path .\05_Split-Shakespeare.psd1 -FunctionsToExport Split-
Shakespeare,Initialize-OutputPath
```

Let's try importing our module by using the manifest and see what happens.

```
# You may have to modify the path below to reflect your local directory
Import-Module .\05_Split-Shakespeare.psm1
Get-Command -Module 05_Split-Shakespeare
Import-Module .\05_Split-Shakespeare.psd1
Get-Command -Module 05_Split-Shakespeare
```

In the code above, we imported our module twice, and listed the commands for each import. When we import our module by referencing the `.psd1` file, you'll see that only the functions we specified for export are listed; we also have proper versioning information. Even though the helper functions like `Split-Books` are no longer exposed, they're still perfectly functional from inside the module, meaning they can still be shared between any additional functions we write that belong to the same module. These "hidden" functions are called `Private Functions`, and we can confirm they still work as intended by invoking `Split-Shakespeare`.

There are many other reasons to use module manifests, some of which will be discussed in later exercises. In the meantime, you can read more about module manifests here.

Exercise 2 - Create a Non-Monolithic Module

Before we discuss the next topic, a quick disclaimer: there's much debate within the PowerShell community on how to organize your modules, and there is no right or wrong way to go about it; it's simply a matter of personal preference. But non-monolithic modules are equally as popular as monolithic versions, and both warrant discussion.

`Monolithic Modules` are fantastic in the fact that they are almost completely self-contained, but they can become unwieldy and difficult to navigate the larger they get. If you have a module with 50 advanced functions in it, it's not uncommon to see line counts exceeding 10,000. In order to avoid this scenario, many people choose to split their functions each into individual files and import them into the `.psm1` file during runtime. This type of architecture is called a `Non-Monolithic Module`.

Let's start by creating a folder with the same name as our PowerShell module file, and then create two folders inside that: one named `Public`, and the other named `Private`. In fact, let's reuse one of the functions in our module to do so: `Initialize-OutputPath`. We can also create another function named `Split-Module` that'll do the heavy lifting for us.

First, we'll add the `Split-Module` function to the end of our 05_Split-Shakespeare module.

```
Function Split-Module
{
    PARAM
    (
        [Parameter(Mandatory = $true)]
        [String]$Name
    )

    # Declare Variables
    $Commands = @()
    $Module   = @()

    $Module = Import-Module $Name -PassThru
    $Module = Import-Module $Module.Path -Force -PassThru

    $Commands = Get-Command -Module $Module.Name

    FOREACH ($Command in $Commands)
    {
        Write-Output "Function $($Command.Name) `n{`n$($Command.Definition)`n}" |
Out-File -FilePath "$($Module.ModuleBase)/$($Command.Name).ps1" -Force
    }
}
```

Now we can reimport our module with the new function, and reuse our custom functions to split up our module into individual files.

```
# Before running this, make sure your PowerShell working directory is the same
folder as the file you're currently reading!
$ModuleName = "05_Split-Shakespeare"
Import-Module ".\$ModuleName.psd1" -Force

# Create output paths using our custom Initialize-OutputPath function
```

```
Initialize-OutputPath -OutputPath $ModuleName\Public
Initialize-OutputPath -OutputPath $ModuleName\Private

# Copy module files into new module directory
Copy-Item -Path "$ModuleName.psm1" -Destination $ModuleName
Copy-Item -Path "$ModuleName.psd1" -Destination $ModuleName

# Split module file using our custom Split-Module function
Set-Location $ModuleName
Split-Module -Name .\05_Split-Shakespeare.psd1
```

If all went well, we should now have a directory structure that looks like this:

```
05_Split-Shakespeare
  ∟ Private\
  ∟ Public\
  ∟ 05_Split-Shakespeare.psd1
  ∟ 05_Split-Shakespeare.psm1
  ∟ Find-BookTitle.ps1
  ∟ Find-BookType.ps1
  ∟ Initialize-OutputPath.ps1
  ∟ Out-Book.ps1
  ∟ Split-Books.ps1
  ∟ Split-Module.ps1
  ∟ Split-Shakespeare.ps1
```

Let's move the functions we want to export (`Initialize-OutputPath`, `Split-Module`, `Split-Shakespeare`) into the `Public` directory, and the rest of the helper functions into the `Private` directory. Our structure should now look like this:

```
05_Split-Shakespeare
  ∟ Private\
    ∟ Find-BookTitle.ps1
    ∟ Find-BookType.ps1
    ∟ Out-Book.ps1
    ∟ Split-Books.ps1
  ∟ Public\
    ∟ Initialize-OutputPath.ps1
    ∟ Split-Module.ps1
    ∟ Split-Shakespeare.ps1
  ∟ 05_Split-Shakespeare.psd1
  ∟ 05_Split-Shakespeare.psm1
```

It looks great, and is easy to navigate, but it's not yet really functional; we still have to edit our module file. Because we've moved our functions into individual files, we don't need to declare them in the `.psm1` file at all. So let's delete all the content in that file and start over. Instead of declaring functions, we'll *dot source* them. We can also make some quality of life improvements, like having the module manifest automatically update it's list of exported functions. Replace the entirety of our newly-copied 05_Split-Shakespeare.psm1 file with the code below.

```
# Get public and private function files
$Public  = @(Get-ChildItem -Path $PSScriptRoot\Public\*.ps1  -ErrorAction
SilentlyContinue)
$Private = @(Get-ChildItem -Path $PSScriptRoot\Private\*.ps1 -ErrorAction
SilentlyContinue)

$Manifest = Get-ChildItem -Path $PSScriptRoot\*.psd1

# Dot source the files
FOREACH ($Import in ($Public + $Private))
{
    TRY
    {
        . $Import.FullName
    }
    CATCH
    {
        Write-Error -Message "Failed to import function $($Import.FullName): $_"
    }
}

# Update module manifest on each subsequent reload with any new .ps1 files that have
been created in the public directory
Update-ModuleManifest -FunctionsToExport $Public.BaseName -Path $Manifest.FullName
```

Now we'll automatically update our manifest based on our file location every single time we run `Import-Module`, and we'll retain the functionality of all our functions, both public and private. Why does this work? As we discussed previously, when we dot-source a script, we're moving functions, variables, aliases, and drives used in it are transferred into the current scope. So by dot-sourcing inside our `.psm1` file, we're changing the scope from the `Script` scope of each `.ps1` file to the `Script` scope of the `.psm1` file, and then finally into the `Local` scope of our session when we run `Import-Module`.

We've made a significant number of changes to our module, and this would be an excellent reason to update the `ModuleVersion` field in our manifest to reflect those changes. Open our new 05_Split-Shakespeare.psd1 file and change the version field to `1.1.0` and save the file (Or run `Update-ModuleManifest` if you prefer using PowerShell!). If you were using any type version control software like Git, this would also be an excellent time to perform a commit to your version history. Keeping track of versioning in your modules not only allows you to see which revisions you're running in your environment, but also allows you run `Import-Module` with the `Version` parameter, which can be extremely useful for development and troubleshooting purposes.

Note: the version number 1.1.0 was chosen because the functionality and structure changes that were made would be something that most would consider a minor change; semantic versioning follows the general rule of `Major.Minor.Patch` in terms of syntax. You can read more about semantic versioning here.