Module 1: Creating advanced functions

Lab A: Converting a command into an advanced function

Scenario

In this exercise, you will convert a command into a parameterized advanced function and test the function.

Task 1: Test an existing command

Test the following command by running it in the PowerShell console:

```
Get-ItemPropertyValue -Path C:\Windows\explorer.exe -Name VersionInfo | Select-Object ProductVersion, FileVersion, CompanyName, FileName
```

Task 2: Identify command values that can be parameterized

Consider the command that you ran in the previous task. Specify the parameter values that should be parameterized so that they can be changed every time the command runs.

Task 3: Create an advanced function

1. Convert the command into an advanced function named `Get-FileVersion`. Include the `[CmdletBinding()]` attribute and a `Param()` block.
2. Define a string parameter named `$FileName` and use the parameter instead of the hard-coded value.

Task 4: Test the advanced function

1. Define the `Get-FileVersion` function in the PowerShell console and after that, run it with the parameter *C:\Windows\Explorer.exe*.

Solutions to Lab A

- **Task 1**: Test an existing command

Run the following command in a PowerShell console:

```
Get-ItemPropertyValue -Path C:\Windows\Explorer.exe -Name VersionInfo | Select-Object
ProductVersion, FileVersion, CompanyName, FileName
```

- **Task 2**: Identify command values that can be parameterized

Notice that the `-Path` parameter can be parameterized in a function

- **Task 3**: Create an advanced function

An example of such function is the following code block:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [string]$FileName
    )

    Get-ItemPropertyValue -Path $FileName -Name VersionInfo | Select-Object ProductVersion,
FileVersion, CompanyName, FileName
}
```

- **Task 4**: Test the advanced function

Copy and run the function block code in a PowerShell console. After that, you can test it with:

```
Get-FileVersion -FileName C:\Windows\Explorer.exe
```

Lab B: Creating a script module

Scenario

In this exercise, you will create a script module.

Task 1: Create a script module

Create a user module named *CorpTools* with the advance function `Get-FileVersion` created in the previous exercise. Save it in the user location that is specified in the first place of the *PSModulePath* environment variable.

Task 2: Test the script module

In a **new** PowerShell console, run the command:

```
Get-FileVersion -FileName C:\Windows\Explorer.exe
```

Task 3: Add verbose output to the script module

1. Change the script module to include **verbose** output that displays the message “Checking `$FileName`”, before querying the file. Save the script.
2. Reload the module.

Task 4: Test the script module

In the same PowerShell window, run the `Get-FileVersion` command enabling verbose output.

> **Question**
>
> What are the advantages of a script module over a regular script?

Solutions to Lab B

- **Task 1**: Create a script module

Execute the following command to see all possible paths for PowerShell modules:

```
$env:PSModulePath.split(';')
```

Create a new folder named *CorpTools* under the first path specified in the *PSModulePath* variable (`Documents\PowerShell\Modules\CorpTools` for PS7) and then create a file named `CorpTools.psm1` in that folder. Copy the function `Get-FileVersion` created in the previous Lab exercise to this file.

Note: The previous action will create a user module. If instead, a system module is needed, the file should be created or copied to `C:\Program Files\PowerShell\Modules` or similar (Administrator rights required). Check the precise folder with the content of the *PSModulePath* variable.

- **Task 2**: Test the script module

Start a **new** PowerShell console, and then run the following command:

```
Get-FileVersion -FileName C:\Windows\Explorer.exe
```

The command should return version information about your the file.

- **Task 3**: Add verbose output to the script module

Edit, with the editor of your choice, the module you just created and add a `Write-Verbose` command. Like this:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [string]$FileName
    )

    Write-Verbose "Checking $FileName"
    Get-ItemPropertyValue -Path $FileName -Name VersionInfo | Select-Object ProductVersion,
FileVersion, CompanyName, FileName
}
```

(added line 7)

To reload the module without reopening the terminal, execute:

```
Import-Module -Name CorpTools -Force
```

- **Task 4**: Test the script module

After reloading the module, execute the cmdlet this way:

```
Get-FileVersion -FileName C:\Windows\Explorer.exe -Verbose
```

Notice the "Checking C:\Windows\Explorer.exe" in the output

Lab C: Defining parameter attributes and input validation

Scenario

In this exercise, you will add parameter and validation attributes to your script module.

Task 1: Mark a parameter as Mandatory

If not already open, open again the *CorpTools* module in the editor of your choice and make the necessary changes to convert the `-FileName` parameter to mandatory.

Task 2: Provide parameter Help messages

Next, add a help message to the `-FileName` parameter

Task 3: Define parameter input validation

Also add a validate pattern to the `-FileName` parameter to only accept *.exe* files using the regular expression "*exe$*"

Task 4: Define a parameter name alias

And finally define a `-ItemName` alias for the `-FileName` parameter

Task 5: Test the modified advanced function

1. Save your changes to the script and reload the module.
2. In a PowerShell window, run the modified `Get-FileVersion` command. Provide the filename *C:\Windows\explorer.exe*.
3. Run `Get-FileVersion` again, providing the filename *C:\Windows\win.ini*. An error should occur.
4. Run `Get-FileVersion` again and do not provide a filename. When prompted, enter !? and confirm the Help message. Then, enter the filename *C:\Windows\notepad.exe*.

Question

When might you use a default parameter value instead of making the parameter mandatory?

Solutions to Lab C

- **Task 1**: Mark a parameter as Mandatory

If not already open, open in the editor of your choice the *CorpTools* module and make the parameter mandatory:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True)]
        [string]$FileName
    )

    Write-Verbose "Checking $FileName"
    Get-ItemPropertyValue -Path $FileName -Name VersionInfo | Select-Object ProductVersion,
FileVersion, CompanyName, FileName
}
```

(added line 4)

- **Task 2**: Provide parameter Help messages

Add a help message for the parameter:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   HelpMessage='Filename')]
        [string]$FileName
    )

    Write-Verbose "Checking $FileName"
    Get-ItemPropertyValue -Path $FileName -Name VersionInfo | Select-Object ProductVersion,
FileVersion, CompanyName, FileName
}
```

- **Task 3**: Define parameter input validation

Add an input validation pattern for the parameter:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   HelpMessage='Filename')]
        [ValidatePattern('exe$')]
        [string]$FileName
    )

    Write-Verbose "Checking $FileName"
    Get-ItemPropertyValue -Path $FileName -Name VersionInfo | Select-Object ProductVersion,
FileVersion, CompanyName, FileName
}
```

- **Task 4**: Define a parameter name alias

Add a *ItemName* alias to the parameter:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   HelpMessage='Filename')]
        [Alias('ItemName')]
        [ValidatePattern('exe$')]
        [string]$FileName
    )

    Write-Verbose "Checking $FileName"
    Get-ItemPropertyValue -Path $FileName -Name VersionInfo | Select-Object ProductVersion,
FileVersion, CompanyName, FileName
}
```

- **Task 5**: Test the modified advanced function

1. Save your modifications to the script
2. Reload the module in the PowerShell window:

```
Import-Module -Name CorpTools -Force
```

3. Run the following code:

```
Get-FileVersion -FileName C:\Windows\Explorer.exe
```

Notice that the command runs correctly.

4. Run the following code

```
Get-FileVersion -FileName C:\Windows\win.ini
```

Review the error message.

5. Run the following code:

```
Get-FileVersion
```

When prompted, enter !? and review the Help message. Then, enter *C:\Windows\Notepad.exe* and press *Enter*.

Review the command output.

Lab D: Writing functions that use multiple objects

Scenario

In this exercise, you will change an existing function so that it enumerates a collection of objects.

Task 1: Enumerate objects in a function

1. Review the existing `Get-FileVersion` function.
2. Change the `$FileName` parameter to accept multiple string values. Change the function to enumerate the filenames and to query each one, one at a time.
3. Save the script and reload the CorpTools module.

Task 2: Test the function

1. In a PowerShell windows run `Get-FileVersion`, providing the filenames *C:\Windows\explorer.exe* and *C:\Windows\notepad.exe*
2. Review the command output.

Solutions to Lab D

- **Task 1**: Enumerate objects in a function

If not already open, open the *CorpTools* module in the editor of your choice and modify it to look like this:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   HelpMessage='One or more filenames')]
        [Alias('ItemName')]
        [ValidatePattern('exe$')]
        [string[]]$FileName
    )

    foreach ($File in $FileName) {
        Write-Verbose "Checking $File"
        Get-ItemPropertyValue -Path $File -Name VersionInfo | Select-Object ProductVersion,
FileVersion, CompanyName, FileName
    }
}
```

In the PowerShell window reload the module:

```
Import-Module -Name CorpTools -Force
```

- **Task 2**: Test the function

In the PowerShell window, execute the following code:

```
Get-FileVersion -FileName C:\Windows\explorer.exe,C:\Windows\Notepad.exe
```

Review the command output

> **Question**
>
> Why did the solution script indent the code that was inside the *ForEach* constructs?

Question

If it is a best practice to use plural names for variables that contain collections, why did the lab use `$FileName` to contain one or more values?

Lab E: Writing functions that accept pipeline input

Scenario

In this exercise, you will change an existing function so that it accepts and uses pipeline input.

Task 1: Define parameters that accept pipeline input

1. Review the existing `Get-FileVersion` function.
2. Change the `-FileName` parameter so that it accepts input from the pipeline by using both *ByValue* and *ByPropertyName*.

Task 2: Change the function to use pipeline input

Change the body of the function to have a **PROCESS** script block.

Save the script and reload the module.

Task 3: Test the function

In a PowerShell console window, run each of the following commands to test the function:

```
Get-FileVersion -FileName C:\Windows\Explorer.exe

"C:\Windows\Notepad.exe","C:\Windows\Explorer.exe" | Get-FileVersion
```

Question

What parameters should accept pipeline input?

Solutions to Lab E

- **Task 1**: Define parameters that accept pipeline input

If not already open, open the *CorpTools* module in the editor of your choice and make the necessary modifications so `-ComputerName` parameter accepts pipeline input:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True,
                   HelpMessage='One or more filenames')]
        [Alias('ItemName')]
        [ValidatePattern('exe$')]
        [string[]]$FileName
    )

    foreach ($File in $FileName) {
        Write-Verbose "Checking $File"
        Get-ItemPropertyValue -Path $File -Name VersionInfo | Select-Object ProductVersion,
FileVersion, CompanyName, FileName
    }
}
```

- **Task 2**: Change the function to use pipeline input

Modify the body of the function to have a **PROCESS** script block:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True,
                   HelpMessage='One or more filenames')]
        [Alias('ItemName')]
        [ValidatePattern('exe$')]
        [string[]]$FileName
    )

    PROCESS {
        foreach ($File in $FileName) {
            Write-Verbose "Checking $File"
            Get-ItemPropertyValue -Path $File -Name VersionInfo | Select-Object ProductVersion,
FileVersion, CompanyName, FileName
        }
    }
}
```

Save the script and reload the module with:

```
Import-Module -Name CorpTools -Force
```

- **Task 3**: Test the function

In a PowerShell console window, run each of the following commands to test the function:

```
Get-FileVersion -FileName C:\Windows\Explorer.exe

"C:\Windows\Notepad.exe","C:\Windows\Explorer.exe" | Get-FileVersion
```

Review the output

Lab F: Producing complex function output

Scenario

In this exercise, you will change an existing function so that it combines data from two sources and produces a custom output object.

Task 1: Save existing command output to a variable

Review the existing `Get-FileVersion` function and make changes to save the command output in a new variable called `$version`.

Task 2: Add more commands to a function

Change the function to include a second and third command:

- The second command should get the file's creation date/time and save the results in `$CreationDate`.
- The third command should get the file's last access date/time and save the results in `$LastAccessDate`.

Task 3: Combine selected properties of command outputs

Combine the following properties into a hash table. Store the hash table in a variable called `$properties` and use the following values:

- For *FileName*, use `$File`.
- For *ProductVersion*, use `$version.ProductVersion`.
- For *FileVersion*, use `$version.FileVersion`.
- For *CreationDate*, use `$CreationDate`.
- For *LastAccessDate*, use `$LastAccessDate`.

Task 4: Produce a custom object

1. Create a new object that uses the properties in `$properties`. Save the object in `$output`.
2. Modify the script to write `$output` to the pipeline.
3. Save the script.
4. Reload the module.

Solutions to Lab F

- **Task 1**: Save existing command output to a variable

If not already open, open the *CorpTools* module in the editor of your choice and modify the function to save the command output in the `$version` variable:

```
# Just showing PROCESS block. Rest of the file omitted
PROCESS {
    foreach ($File in $FileName) {
        Write-Verbose "Checking $File"
        $version = Get-ItemPropertyValue -Path $File -Name VersionInfo | Select-Object
ProductVersion, FileVersion, CompanyName, FileName
    }
}
```

- **Task 2**: Add more commands to a function

Add two commands to retrieve creation time and last access time and save their output to the `$CreationDate` and `$LastAccessDate` variables:

```
# Just showing PROCESS block. Rest of the file omitted
PROCESS {
    foreach ($File in $FileName) {
        Write-Verbose "Checking $File"
        $version = Get-ItemPropertyValue -Path $File -Name VersionInfo | Select-Object
ProductVersion, FileVersion, CompanyName, FileName
        $CreationDate = Get-ItemProperty -Path $File | Select CreationTime
        $LastAccessDate = Get-ItemProperty -Path C:\Windows | Select LastAccessTime
    }
}
```

[Continue on the next page...]

- **Task 3**: Combine selected properties of command outputs

Combine the information from the different file property values in a single hash table:

```
# Just showing PROCESS block. Rest of the file omitted
PROCESS {
    foreach ($File in $FileName) {
        Write-Verbose "Checking $File"
        $version = Get-ItemPropertyValue -Path $File -Name VersionInfo | Select-Object
ProductVersion, FileVersion, CompanyName, FileName
        $CreationDate = Get-ItemProperty -Path $File | Select -ExpandProperty CreationTime
        $LastAccessDate = Get-ItemProperty -Path C:\Windows | Select -ExpandProperty
LastAccessTime

        $properties = @{'FileName'       = $File;
                        'ProductVersion' = $version.ProductVersion;
                        'FileVersion'    = $version.FileVersion;
                        'CreationDate'   = $CreationDate;
                        'LastAccessDate' = $LastAccessDate;
        }
    }
}
```

[Continue on the next page...]

- **Task 4**: Produce a custom object

Produce a custom object that uses the properties in the hash table:

```
Function Get-FileVersion {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True,
                   HelpMessage='One or more filenames')]
        [Alias('ItemName')]
        [ValidatePattern('exe$')]
        [string[]]$FileName
    )

    PROCESS {
        foreach ($File in $FileName) {
            Write-Verbose "Checking $File"
            $version = Get-ItemPropertyValue -Path $File -Name VersionInfo | Select-Object
ProductVersion, FileVersion, CompanyName, FileName
            $CreationDate = Get-ItemProperty -Path $File | Select -ExpandProperty CreationTime
            $LastAccessDate = Get-ItemProperty -Path C:\Windows | Select -ExpandProperty
LastAccessTime

            $properties = @{'FileName'       = $File;
                            'ProductVersion' = $version.ProductVersion;
                            'FileVersion'    = $version.FileVersion;
                            'CreationDate'   = $CreationDate;
                            'LastAccessDate' = $LastAccessDate;
            }
            $output = New-Object -TypeName PSObject -Property $properties
            Write-Output $output
        }
    }
}
```

In the PowerShell window reload the module:

```
Import-Module -Name CorpTools -Force
```

[Continue on the next page...]

- **Task 5**: Test the function

In the PowerShell window run the following command:

```
Get-FileVersion -FileName C:\Windows\Explorer.exe, C:\Windows\Notepad.exe
```

And review the output.

Lab G: Documenting functions by using comment-based Help

Scenario

In this exercise, you will change an existing function so that it includes comment-based Help.

Task 1: Add comment-based Help

1. Review the existing `Get-FileVersion` function.
2. Add comment-based Help to the function. Include at least the following sections:

- **SYNOPSIS**
- **DESCRIPTION**
- **PARAMETER** FileName
- **EXAMPLE** with two examples

3. Save the script.
4. Reload the *CorpTools* module.

Task 2: Test the function

In a PowerShell window, run the following command:

```
Get-Help Get-FileVersion -full
```

Solutions to Lab G

- **Task 1**: Add comment-based Help

Add and fill the help sections like this:

```
Function Get-FileVersion {
<#
.SYNOPSIS
Retreives File and Product version, Creation date and Last Access Date from one or more .exe
files.
.DESCRIPTION
This command retrieves specific information from each file defined in the input. The command
will only work with .exe files.
.PARAMETER FileName
One or more filenames, as strings. Use full path is necessary. This parameter accepts pipeline
input. Filenames must end in ".exe".
.EXAMPLE
 Get-ChildItem *.exe | Get-FileVersion
This example assumes that there are .exe files in the current directory, and will retrieve
information from each file listed.
.EXAMPLE
 Get-FileVersion -FileName C:\Windows\explorer.exe
This example retrieves information from one file.
#>

# Rest of the file omitted...
```

Save the script and reload it with:

```
Import-Module -Name CorpTools -Force
```

- **Task 2**: Test the function

In a PowerShell window, enter the following command, and then press Enter:

```
Get-Help Get-FileVersion -Full
```

Question

When displaying the full Help for your command, Windows PowerShell displays additional information about the `-FileName` parameter. For example, it listed the fact that the parameter is required instead of being optional. Where did that information come from?

Lab H: Supporting -WhatIf and -Confirm

Scenario

In this exercise, you will change a function so that it supports the `-WhatIf` and `-Confirm` parameters.

Task 1: Declare support for ShouldProcess

Although the `Get-FileVersion` doesn't modify anything, as an exercise you will declare support for **ShouldProcess** in it. Specify **ConfirmImpact** as *Medium*.

Task 2: Support ShouldProcess

In the script, identify the lines that do the actual job and implement support for **ShouldProcess**.

Save the script and reload the *CorpTools* module.

Task 3: Test the function

In a PowerShell window, run the following code:

```
Get-FileVersion -FileName C:\Windows\explorer.exe,C:\Windows\Notepad.exe
```

Verify that a **Whatif** message displays.

> **Question**
>
> Why would you implement **ShouldProcess** only around the fewest number of possible commands?

Solutions to Lab H

- **Task 1**: Declare support for **ShouldProcess**

If not already open, open the module in the editor of your choice and declare support for *ShouldProcess* in the `Get-FileVersion` function:

```
Function Get-FileVersion {
# Help block omitted...
    [CmdletBinding(SupportsShouldProcess=$True,ConfirmImpact='Medium')]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True,
                   HelpMessage='One or more filenames')]
        [Alias('ItemName')]
        [ValidatePattern('exe$')]
        [string[]]$FileName
    )
# Rest of the file omitted...
```

[Continue on the next page...]

- **Task 2**: Support ShouldProcess

Add support for **ShouldProcess** in the body of the function:

```
Function Get-FileVersion {
# Help section omitted...
    [CmdletBinding(SupportsShouldProcess=$True,ConfirmImpact='Medium')]
    Param(
        [Parameter(Mandatory=$True,
                   ValueFromPipeline=$True,
                   ValueFromPipelineByPropertyName=$True,
                   HelpMessage='One or more filenames')]
        [Alias('ItemName')]
        [ValidatePattern('exe$')]
        [string[]]$FileName
    )

    PROCESS {
        foreach ($File in $FileName) {
            if ($PSCmdlet.ShouldProcess($File)) {
                Write-Verbose "Checking $File"
                $version = Get-ItemPropertyValue -Path $File -Name VersionInfo | Select-Object
ProductVersion, FileVersion, CompanyName, FileName
                $CreationDate = Get-ItemProperty -Path $File | Select -ExpandProperty
CreationTime
                $LastAccessDate = Get-ItemProperty -Path C:\Windows | Select -ExpandProperty
LastAccessTime

                $properties = @{'FileName'       = $File;
                                'ProductVersion' = $version.ProductVersion;
                                'FileVersion'    = $version.FileVersion;
                                'CreationDate'   = $CreationDate;
                                'LastAccessDate' = $LastAccessDate;
                }
                $output = New-Object -TypeName PSObject -Property $properties
                Write-Output $output
            }
        }
    }
}
```

Save the module script and reload it with:

```
Import-Module -Name CorpTools -Force
```

- **Task 3**: Test the function

In a PowerShell window, run the following code:

```
Get-FileVersion -FileName C:\Windows\explorer.exe,C:\Windows\Notepad.exe
```

Verify that **Whatif** message/s display.