PowerShell Toolmaking

Create shareable tools that other people will want to use.

Goals & Expectations

Goals

- Quick review of the fundamentals
- Turn one-liners and scripts into advanced functions
- Anatomy of a function
  - Pipelining, validation, tab completion
- Create user documentation using Comment-based help
- Package functions into structured modules
  - Public vs. Private functions
  - Multi-file vs. monolithic
- Command design principles
- Publish modules to a shareable PSRepository

Expectations

- Some experience writing PowerShell scripts
- VS Code with PowerShell extension
- PowerShell 7 (or 5.1)

This is an interactive session with lots of demos. We will be using PowerShell 7.5 (or higher) and Visual Studio Code (VS Code) with the PowerShell extension on Windows 11. The principles covered apply to Windows PowerShell 5.1 and PowerShell 7 running on macOS or Linux.

Feel free to follow along with the presentation on your own computer.

PowerShell fundamentals

- Execution policy overview
  - Security feature – not a security boundary
  - Windows-only
- Command types and their precedence
  - Alias
  - Function
  - Cmdlet
  - External executable files
- Scripts & functions
  - Purpose
  - Batch scripts vs. tools
  - Scriptblocks
  - Profiles and dot-sourcing
- Variable scopes
  - Global
  - Local
  - Script
  - Private

Execution Policy

PowerShell's execution policy is a safety feature that controls the conditions under which PowerShell loads configuration files and runs scripts. This feature helps prevent the execution of malicious scripts. This feature is only available on Windows platforms. Execution policy is ignored on macOS and Linux.

The default policy setting is `RemoteSigned`. Under this policy, script files downloaded from an untrusted network require a digital signature from a trusted publisher. Scripts created and accessed locally don't require digital signatures. You can also run unsigned scripts that are downloaded from the internet, if the scripts are unblocked using the `Unblock-File` cmdlet.

The policy setting can be managed by your IT organization using Group Policies. To check the current policy on your machine, run the `Get-ExecutionPolicy` cmdlet.

```
PS> Get-ExecutionPolicy
RemoteSigned
```

Before you can run scripts, you must change the policy to a value that allows for script execution. To make this change, run the `Set-ExecutionPolicy` cmdlet in an elevated PowerShell session.

```
PS> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

Further reading

- about_Execution_Policies
- PowerShell security features

Command precedence

When a PowerShell session includes more than one command that has the same name, PowerShell determines which command to run using the following rules. You can run any executable command using its full path. As a security feature, PowerShell doesn't run executable commands, including PowerShell scripts and native commands, unless the command is in a path listed in the `$Env:PATH` environment variable.

If you don't specify a path, PowerShell uses the following precedence order when it runs commands.

1. Alias
2. Function
3. Cmdlet
4. External executable files (including PowerShell script files)

Example

Consider the scenario where an alias and a function are created with the same name as an executable file.

```
PS> Set-Alias -Name ipconfig -Value Get-ChildItem
PS> function ipconfig { Write-Host "This is a fake ipconfig function" }
PS> Get-Command ipconfig -All

CommandType Name                     Version         Source
----------- ----                     -------         ------
Alias       ipconfig -> Get-ChildItem
Function    ipconfig
Application ipconfig.exe             10.0.26100.6725 C:\WINDOWS\system32\ipconfig.exe
```

`Get-Command` shows the commands with the name `ipconfig` in the order of execution precedence. When you run `ipconfig`, it run the command mapped to the alias. After deleting the alias, it runs the function. After deleting the function, it runs the application.

```
PS> ipconfig /all
Get-ChildItem: Cannot find path 'D:\all' because it does not exist.

PS> del alias:ipconfig
PS> ipconfig /all
This is a fake ipconfig function

PS> del function:ipconfig
PS> ipconfig

Windows IP Configuration

Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . :
   IPv4 Address. . . . . . . . . . . : 192.168.1.170
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
```

Further reading

- about_Command_Precedence
- about_Aliases
- about_Functions
- about_Path_Syntax
- about_Providers
- about_Alias_Provider
- about_Function_Provider
- Get-Command

Scriptblocks, functions, & scripts

A **scriptblock** is a collection of statements or expressions that can be used as a single unit. The collection of statements enclosed in braces (`{ }`). A script block can return values and accept parameters and arguments.

A **function** is a named scriptblock. To run a function, you type the name of the function.

PowerShell defines two kinds of functions:

- A **function** is a block of code that can be called by name. It can take input and return output. Functions are defined using the `function` keyword.
- A **filter** is a type of function designed to process data from the pipeline. Filters are defined using the `filter` keyword.

A **script** is a plain text file that contains one or more commands, expressions, scriptblocks, or functions. PowerShell scripts have a `.ps1` file extension.

Profile scripts

A PowerShell profile is a script that runs when PowerShell starts. You can use the profile as a startup script to customize your environment. You can add commands, aliases, functions, variables, modules, PowerShell drives and more.

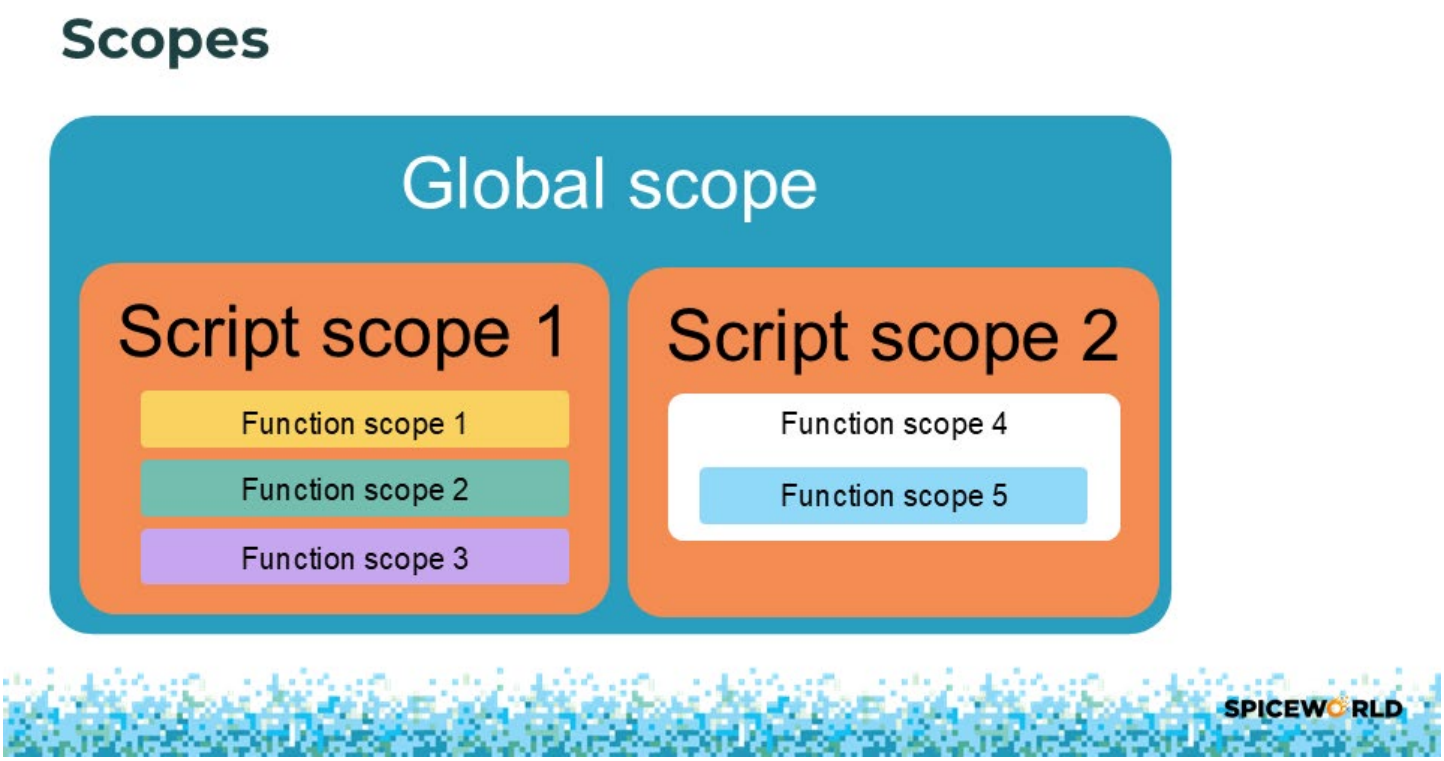
The automatic variable `$PROFILE` is a string with four value properties. Each property defines the location and purpose of the profile. The `CurrentUserCurrentHost` property is the default profile. PowerShell doesn't create these files automatically. You must create them as needed. All profiles are executed in the order listed in the example. The `CurrentUserCurrentHost` profile is the last to execute, so it can override any changes made in the previous profiles.

```
PS> $PROFILE | Select-Object *

AllUsersAllHosts       : C:\Program Files\PowerShell\7-preview\profile.ps1
AllUsersCurrentHost    : C:\Program Files\PowerShell\7-
preview\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\sewhee\Documents\PowerShell\profile.ps1
CurrentUserCurrentHost :
C:\Users\sewhee\Documents\PowerShell\Microsoft.PowerShell_profile.ps1
Length                 : 69
```

At startup, PowerShell runs the profile script in the Global scope. This is equivalent to dot-sourcing the script file. That means that any variables, aliases, functions, etc. defined at the top level of the script are added to your PowerShell session.

Further reading

- about_Script_Blocks
- about_Functions
- about_Scripts
- about_Profiles
- Make your PowerShell profile work cross-platform - Sean on IT



Scopes

PowerShell protects access to variables, aliases, functions, and PowerShell drives (PSDrives) by limiting where they can be read and changed. PowerShell uses scope rules to ensure that you don't make unintentional changes to items in other scopes.

The following are the basic rules of scope:

- Scopes may nest. An outer scope is referred to as a parent scope. Any nested scopes are child scopes of that parent.
- An item is visible in the scope that it was created and in any child scopes, unless you explicitly make it private.
- Using scope modifiers, you can declare variables, aliases, functions, and PowerShell drives for a scope outside of the current scope.
- An item that you created within a scope can be changed only in the scope in which it was created, unless you explicitly specify a different scope.
- When code running in a runspace references an item, PowerShell searches the scope hierarchy, starting with the current scope and proceeding through each parent scope.
    - If the item isn't found, a new item is created in the current scope.
    - If it finds a match, the value of the item is retrieved from the scope where is was found.
    - If you change the value, the item is copied to the current scope so that the change only affects the current scope.

- If you explicitly create an item that shares its name with an item in a different scope, the original item might be hidden by the new item, but it isn't overridden or changed.

PowerShell defines the following named scopes:

- **Global**: The scope that's in effect when PowerShell starts or when you create a new session or runspace. Variables and functions that are present when PowerShell starts, such as automatic variables and preference variables, are created in the global scope. The variables, aliases, and functions in your PowerShell profiles are also created in the global scope. The global scope is the root parent scope in a runspace.
- **Local**: The current scope. The local scope can be the global scope or any other scope.
- **Script**: The scope that's created while a script file runs. The commands in the script run in the script scope. For the commands in a script, the script scope is the local scope.

The default scope for scripts is the script scope. The default scope for functions and aliases is the local scope, even if they're defined in a script.

A variable, alias, or function name can include any one of the following optional scope modifiers:

- **Global**: - Specifies that the name exists in the **Global** scope.
- **Local**: - Specifies that the name exists in the **Local** scope. The current scope is always the **Local** scope.
- **Private**: - Specifies that the name is **Private** and only visible to the current scope. **Note** This isn't a scope. It's an **option** that changes the accessibility of an item outside of the scope in which it's defined.
- **Script**: - Specifies that the name exists in the **Script** scope. **Script** scope is the nearest ancestor script file's scope or **Global** if there is no nearest ancestor script file.
- **Using**: - Used to access variables defined in another scope while running in remote sessions, background jobs, or thread jobs.
- **<variable-namespace>** - A modifier created by a PowerShell **PSDrive** provider. For example: `Alias:`, `Env:`, `Function:`, `Variable:`.

Further reading

- about_Scopes
- Using scopes in PowerShell with Steven Judd - PowerShell Wednesday

Anatomy of a function

A **function** is a block of code that can be called by name. It can take input and return output. Functions are defined using the `function` keyword. PowerShell statements are grouped into one of four different labeled script blocks. These script blocks are labeled using the keywords `begin`, `process`, `end`, and `clean`. If you don't use these keywords, PowerShell puts the statements in the appropriate code block.

A **filter** is a type of function designed to process data from the pipeline. Filters are defined using the `filter` keyword. To simplify the syntax for filters, omit the script block label (`begin`, `process`, `end`, or `clean`). PowerShell puts the statements in the `process` block. You can use any of the other blocks in a filter, but the intent was to provide a shorthand way of defining a function that has the sole purpose of processing each object in the pipeline.

- The `begin` block provides (optional) one-time preprocessing for the function. PowerShell runs the code in this block once for each instance of the function in the pipeline.
- The `process` block provides record-by-record processing for the function. You can use a process block without defining the other blocks. The process block is executed for each item received via the pipeline.
- The `end` block to provide optional one-time post-processing for the function.
- The `clean` block (added in PowerShell 7.3) provide a convenient way to clean up resources, such as connection or temp files, created by the function that are no longer needed. It's semantically similar to a `finally` block but scoped to the function. Resource cleanup is enforced for the following scenarios:

  1. when the pipeline execution finishes without terminating error
  2. when the pipeline execution is interrupted due to terminating error
  3. when the pipeline is truncated, for example: Select-Object -First
  4. when the pipeline is stopped by `<Ctrl+C>` or `StopProcessing()`

Further reading

- *Input processing method* section of `about_Functions`.

Advanced Functions

Advanced functions use the `[CmdletBinding()]` attribute and allow you create functions that behave like cmdlets. Advanced functions make it easier to create cmdlets without having to create compiled C# binary cmdlets.

The parameters of the function are variables declared in the `param()` statement. You can use the optional `[Parameter()]` attribute alone or in combination with parameter attributes. If you use the `[Parameter()]` attribute, PowerShell applies the `[CmdletBinding()]` attribute automatically, even if you didn't specify it in the function.

The `[Parameter()]` attribute allows you to set values for several properties. For example, these properties can define the position of argument values on the command line, enable support for pipeline input, and define parameter set names.

```
function Test-ValueFromPipelineByPropertyName {
  [OutputType([string])]
  param(
      [Parameter(Mandatory, ValueFromPipelineByPropertyName, Position=0)]
      [string[]]$ComputerName
  )

  process {
    "Received ComputerName = '$ComputerName'"
  }
}
```

Further reading

- about_Functions_Advanced
- about_Functions_CmdletBindingAttribute
- about_Functions_Advanced_Methods

Pipeline input

A pipeline is a series of commands connected by pipeline operators (`|`). The output of the first command can be sent for processing as input to the second command. And that output can be sent to yet another command. The result is a complex command chain or *pipeline* that's composed of a series of simple commands.

Function parameters can accept pipeline input in two different ways:

- **ByValue**: The parameter accepts values that match the expected .NET type or that can be converted to that type. For example, the `-Name` parameter of `Start-Service` accepts pipeline input by value. It can accept string objects or objects that can be converted to strings.
- **ByPropertyName**: The parameter accepts input only when the input object has a property of the same name as the parameter. For example, the `-Name` parameter of `Start-Service` can accept objects that have a `Name` property. You can also match object properties by name to the *alias* of a parameter. To list the properties of an object, pipe it to `Get-Member`.

Further reading

- about_Pipelines
- Understanding & Troubleshooting PowerShell Parameter Binding - Sean on IT
- Visualize parameter binding

Parameter validation

PowerShell uses validation attributes to test parameter input values. If the parameter values fail the test, PowerShell throws an error and the function isn't executed.

You can also use the validation attributes to:

- Restrict the values allowed for input
  - `[ValidateSet()]`
  - `[ValidateRange()]`
  - `[ValidateNotNull()]`
  - `[ValidateNotNullOrEmpty()]`
  - `[ValidateNotNullOrWhiteSpace()]`
  - `[AllowNull()]`
  - `[AllowEmptyString()]`
- Restrict the values by length or pattern
  - `[ValidateLength()]`
  - `[ValidateCount()]`
  - `[AllowEmptyCollection()]`
  - `[ValidatePattern()]`
  - `[ValidateScript()]`

Further reading

- about_Functions_Advanced_Parameters
- about_Functions_OutputTypeAttribute

Argument completion

Argument completers provide tab completion of parameter values. The possible values are calculated at runtime when the user presses the Tab key after the parameter name. There are several ways to define an argument completer for a parameter.

- `[ValidateSet()]` attribute
- Using `enum` typed parameters
- `[ArgumentCompletions()]` attribute
- `[ArgumentCompleter()]` attribute
- Create class-based argument completer methods
- `Register-ArgumentCompleter` command

Using the `Register-ArgumentCompleter` command is the most common way to add completers. This command also allows you to create completers for native commands.

The script block must accept the following parameters in the order specified below. The names of the parameters aren't important because PowerShell passes in the values by position.

- `$commandName` (Position 0, **String**) - This parameter is set to the name of the command for which the script block is providing tab completion.
- `$parameterName` (Position 1, **String**) - This parameter is set to the parameter whose value requires tab completion.
- `$wordToComplete` (Position 2, **String**) - This parameter is set to value the user has provided before they pressed Tab. Your script block should use this value to determine tab completion values.
- `$commandAst` (Position 3, **CommandAst**) - This parameter is set to the Abstract Syntax Tree (AST) for the current input line.
- `$fakeBoundParameters` (Position 4 **IDictionary**) - This parameter is set to a hashtable containing the `$PSBoundParameters` for the cmdlet, before the user pressed Tab.

The following examples adds tab completion of values for the `-Id` parameter of the `Set-TimeZone` command. The scriptblock get a list of time zone Ids and returns the matching value wrapped in single quotes.

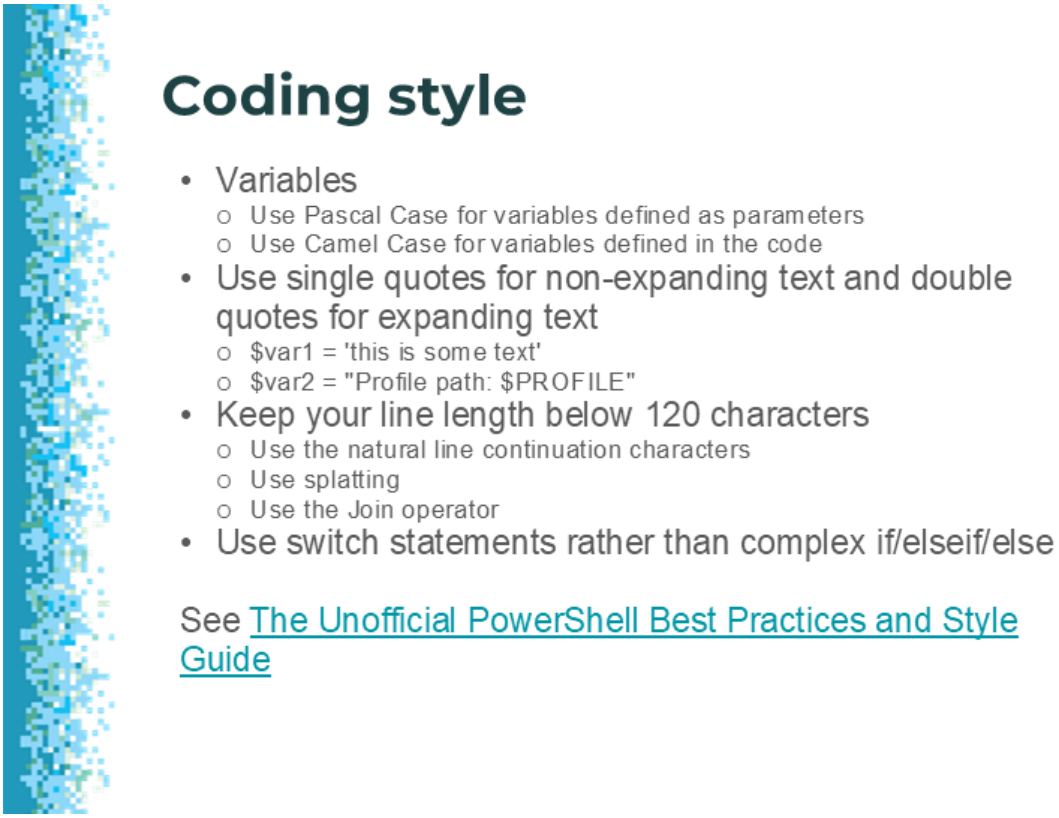
```
$sb = {
    param(
        $commandName,
        $parameterName,
        $wordToComplete,
        $commandAst,
        $fakeBoundParameters
    )

(Get-TimeZone -ListAvailable).Id | Where-Object {
        $_ -like "$wordToComplete*"
    } | ForEach-Object {
        "'$_'"
    }
}

Register-ArgumentCompleter -CommandName Set-TimeZone -ParameterName Id -ScriptBlock
$sb
```

If you want to create an argument completer for a native command, the scriptblock parameters are different. Refer to the documentation for examples. Many newer CLI tools (like the GitHub CLI) have built-in argument completer scripts for several shells. For example, you can add the following line to your PowerShell profile script.

```
Invoke-Expression -Command $(gh completion -s powershell | Out-String)
```

Further reading

- about_Functions_Argument_Completion



Formatting your code

Follow a standard set of code-style habits so that your code is easy to read, understood, and maintain. Most of the PowerShell community has adopted the recommendations of The Unofficial PowerShell Best Practices and Style Guide.

Maintain Consistency in Layout

Rules about indentation, line length, and capitalization are about consistency across code bases. It's easier to read and understand code when it looks familiar and you're not being distracted by details. Since the goal is consistency, you should always abide by any style rules that are in place on the project you are contributing to.

Capitalization Conventions

PowerShell is **not** case sensitive, but we follow capitalization conventions to make code easy to read. They are based on the capitalization conventions Microsoft created for the .NET framework.

Terminology

- lowercase - all lowercase, no word separation
- UPPERCASE - all capitals, no word separation
- PascalCase - capitalize the first letter of each word
- camelCase - capitalize the first letter of each word *except* the first.

PowerShell uses PascalCase for *all* public identifiers: module names, function or cmdlet names, class, enum, and attribute names, public fields or properties, global variables and constants, etc. In fact, since

the *parameters* to PowerShell commands are actually *properties* of .NET classes, even parameters use PascalCase rather than camelCase.

PowerShell language keywords are written in lower case, like `foreach` and `dynamicparam`, as well as operators like `-eq` and `-match`. The keywords in comment-based help are written in UPPERCASE to make them easier to read among the dense prose of documentation.

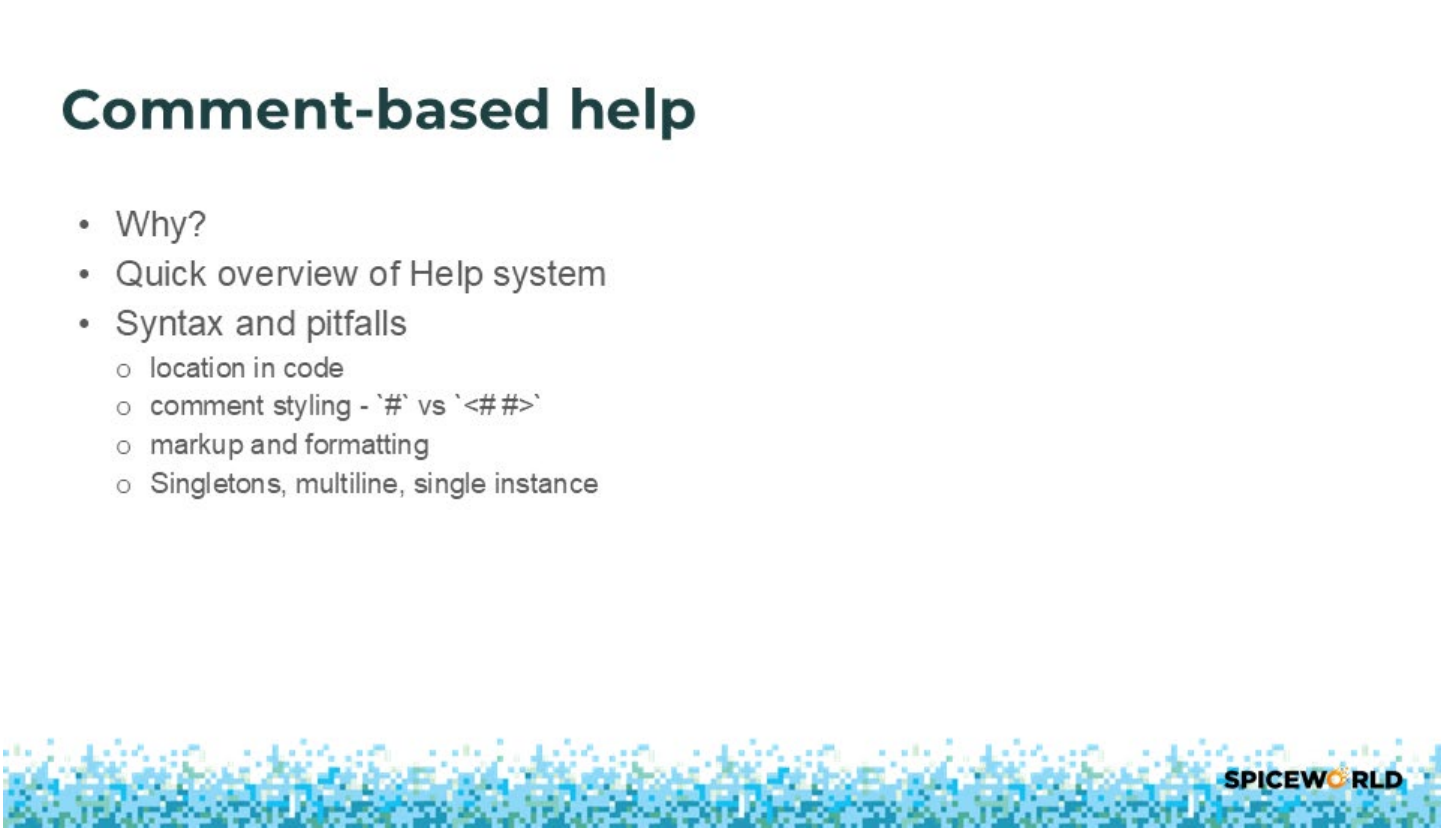
One True Brace Style

The "One True Brace Style" variant to K&R requires that every braceable *statement* should have the opening brace on the *end of a line*, and the closing brace at the *beginning of a line*. There is one notable exception when passing small scriptblocks to parameters, put the entire statement on a single line.

```
enum Color {
    Black,
    White
}

function Test-Code {
    [CmdletBinding()]
    param (
        [int]$ParameterOne
    )
    end {
        if (10 -gt $ParameterOne) {
            "Greater"
        } else {
            "Lesser"
        }
    }
}

# An Exception case:
Get-ChildItem | Where-Object { $_.Length -gt 10mb }
```

Further reading

- Code-Layout-and-Formatting.md
- Documentation-and-Comments.md
- Function-Structure.md



Helping your users

You can write comment-based help content for functions and scripts by using special help comment keywords.

The `Get-Help` cmdlet displays comment-based help in the same format in which it displays the cmdlet help content that are generated from XML files.

Comment-based help for a function can appear in one of three locations:

- At the beginning of the function body.
- At the end of the function body.
- Before the function keyword. There can't be more than one blank line between the last line of the function help and the function keyword.

Autogenerated content

The PowerShell Help system automatically creates help content for every script or function. The following examples shows the autogenerated help for the `Test-ValueFromPipelineByPropertyName` function defined in a previous example. The PowerShell Help system automatically created all the content shown in this example.

```
PS> Get-Help Test-ValueFromPipelineByPropertyName -Full

NAME
    Test-ValueFromPipelineByPropertyName

SYNTAX
    Test-ValueFromPipelineByPropertyName [-ComputerName] <string[]>
[<CommonParameters>]

PARAMETERS
    -ComputerName <string[]>

        Required?                    true
        Position?                    0
        Accept pipeline input?       true (ByPropertyName)
        Parameter set name           (All)
        Aliases                      None
        Dynamic?                     false
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (https://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    System.String[]

OUTPUTS
    System.String

ALIASES
    None

REMARKS
    None
```

Comment-base help keywords

Using comment-based help keywords, you can add descriptions, examples, and links to related information. To create Comment-based help content, you can use either single-line or block style comments.

```
# .<help keyword>
# <help content>

<#
    .<help keyword>
    <help content>
#>
```

The most commonly used keywords are:

- `.SYNOPSIS`
- `.DESCRIPTION`
- `.PARAMETER`
- `.EXAMPLE`
- `.INPUTS`
- `.NOTES`
- `.OUTPUTS`
- `.LINK`

To ensure the help comments are associated with the correct function, you must put the comments in one of the following locations:

- At the beginning of the function body. This is the preferred location.
- At the end of the function body.
- Before the Function keyword. When the function is in a script or script module, there can't be more than one blank line between the last line of the comment-based help and the Function keyword. Otherwise, Get-Help associates the help with the script, not with the function.

The PowerShell help system also supports conceptual ("about") Help articles. These articles describe the module and other conceptual topics. The filename must use the `about_<name>.help.txt` format, such as `about_MyModule.help.txt`. To work properly with the help system, the files must be formatted correctly and included with the module in a specific folder structure. For more information, see Writing Help for PowerShell Modules.

Further reading

- about_Comment_Based_Help
- Writing Comment-Based Help Topics
- Writing Help for PowerShell Modules

Module design

Structure
- Module manifest
- Public vs private functions
- Monolithic vs multiple files
- Templating (Plaster or Stucco)

Other resources
- Help file
- Format files (intermediate)
- Type files (advanced)
- Other data files (as needed)

Module structure

The simplest way to create a module is to rename your `.PS1` file to a `.PSM1` file. However, to publish your module to a PSRepository (like the PowerShell Gallery) you must create a module manifest (`.PSD1` file). Module manifests are covered in more detail later in this presentation.

Monolithic vs. mult-file design

In the PowerShell community, you will find two common approaches for structuring modules.

In the *monolithic* model, all classes and functions are contained in a single `.PSM1` file. In a *multi-file* model, each function is in its own `.PS1` file. The `.PSM1` file dot-sources each `.PS1` file when the module is imported.

Monolithic model	Multi-file model
The module loads faster and there is no problem with load order dependencies.	The module loads slower. You must manage the order that files are dot-source. Loading class definitions is more difficult.
Monolithic files can grow large and be harder to maintain. Testing incremental changes is more difficult.	Separate files are easier to maintain and unit test.
Example: sdwheeler.EssentialUtils module on GitHub	Example: MrInspector module on GitHub

There is a third, hybrid approach that is becoming more popular. In the hybrid approach, you create separate `.PS1` file for every function. Then you use a build script to compose the file monolithic file and create an installable module package. For an example, see the Documentarian modules on GitHub.

Choose the model that works best for you and meets the needs of the project. Use templating tools to scaffold your new projects.

Templating tools

Templating tools are used to create scaffolding for new projects. The template creates a base set of files for the module that you can use to get started with development. They can also create folder structures to help you organize your files and install configuration files for common development tools such as Git/GitHub, VS Code, and PSScriptAnalyzer. Using templates improves consistency across your development projects.

Plaster is a template-based project generator written in PowerShell. Its purpose is to streamline the creation of PowerShell module projects, Pester tests, DSC configurations, and more. **Stucco** is an opinionated Plaster template for building high-quality PowerShell modules.

Project links

- PowerShellOrg/Plaster: Plaster is a template-based file and project generator written in PowerShell.
- devblackops/Stucco: An opinionated Plaster template for high-quality PowerShell modules

YouTube videos

- The Art & Craft of Module Development using Stucco with Josh Hendricks | GNVPSUG
- Modern PowerShell module Development - Gael Colas - PSConfEU 2024

Including other resources

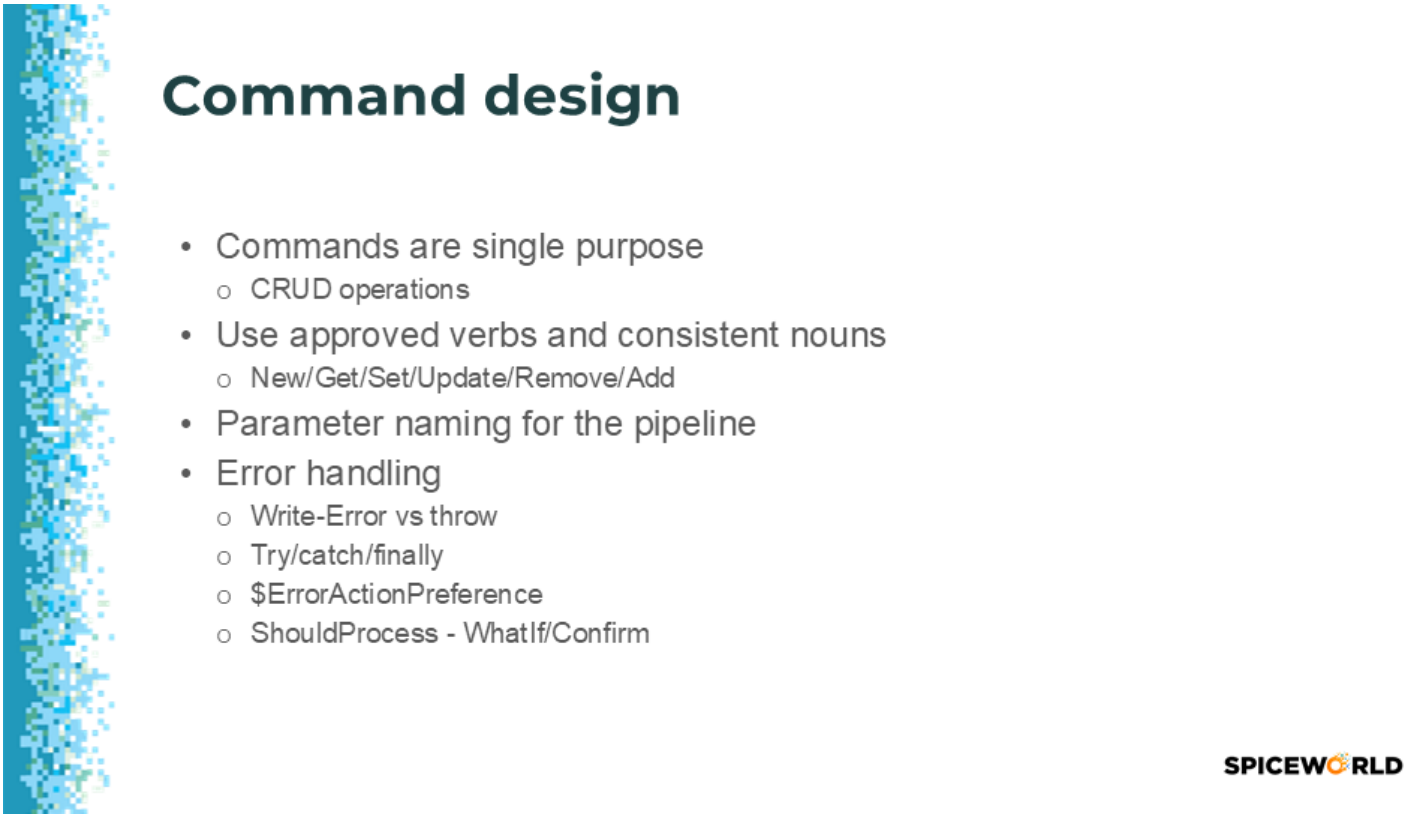
You can also include other resources in your module. For example:

- Help files
- Format (`format.ps1xml`) files – these XML files define how output is presented in the console. For more information, see about_Format.ps1xml.
- Type (types.ps1.xml) files – these XML files can be used to extend existing types and define default properties to be displayed. For more information, see about_Types.ps1xml.
- Other data files (as needed) – Examples include README files, data files used by commands in the module, example scripts, etc.

As mentioned previously, the Help files must go in specific folders. The rest of the files can go anywhere in the folder as long as your module code knows where to find them.

Further reading

- about_Modules
- about_Module_Manifests



Command design

Best practices

- Limit functions to a single task.
  - Easier to write, test, and troubleshoot
  - Allows for composition of pipelines and scripts
  - Think CRUD
- Use approved verbs
  - Avoids noisy warning messages
  - More predictable for users
    - PowerShell becomes just *Shell* if your users have to read the docs before they can use your code.
  - See Approved Verbs for PowerShell Commands
- Prefix your nouns
  - Use your initials or your product or company name (or abbreviation). Pick the one that's least likely to change.
  - For example, a command for BackupExec
    - Use `Get-BEJob` since `Get-Job` is already taken by PowerShell
- Design for pipeline support
  - Use common parameter names
  - Strongly type parameters
  - Align object property names to parameter names or aliases

Further reading

- Required Development Guidelines
- Strongly Encouraged Development Guidelines
- Advisory Development Guidelines

Error handling

A terminating error stops a statement from running. If PowerShell does not handle a terminating error in some way, PowerShell also stops running the function or script using the current pipeline.
Use `try`, `catch`, and `finally` blocks to handle terminating errors in scripts.

The `$ErrorActionPreference` variable controls how PowerShell responds to a non-terminating error. By default, the value is set to **Continue**.

```
PS> $ErrorActionPreference
Continue
```

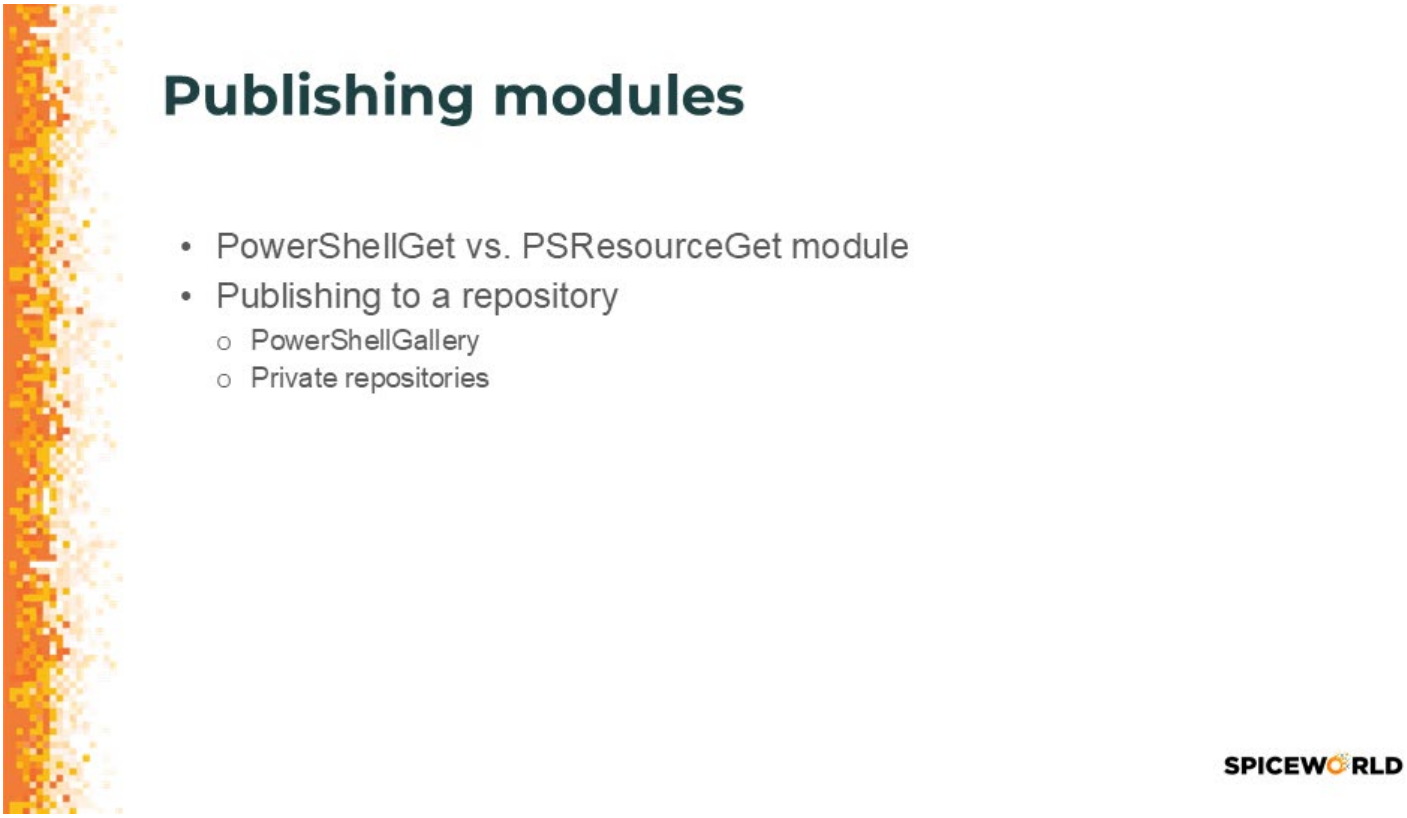
Most PowerShell cmdlets create non-terminating errors. You can change the error behavior of a single command using the `-ErrorAction` common parameter.

The `Write-Error` cmdlet can be used to write error messages to the **Error** stream. However, it doesn't create terminating errors. If you want a `try-catch` block to handle errors, you need to change the `$ErrorActionPreference` variable to **Stop**. If you change the `$ErrorActionPreference` variable inside a script or function, the value and behavior is scoped to where you made the change. When the script or function exits, the value of the variable returns to the current value of the global scope.

For an example of using `try-catch`, see the Show-Redirects function in my GitHub repository.

Further reading

- about_Try_Catch_Finally
- about_Throw
- about_Trap
- Write-Error
- about_CommonParameters
- about_Preference_Variables
- about_Output_Streams



PowerShell package managers

PowerShell come with two package management tools:

- The **Microsoft.PowerShell.PSResourceGet** module - shipped originally in PowerShell 7.4.0
- The **PowerShellGet** and **PackageManagement** modules - shipped originally in Windows PowerShell 5.0

These modules contain cmdlets for discovering, installing, updating, and publishing PowerShell packages from the PowerShell Gallery. These packages can contain artifacts such as Modules, DSC Resources, and Scripts. The **Microsoft.PowerShell.PSResourceGet** module replaces the **PowerShellGet** and **PackageManagement** modules.

PSResourceGet can be installed and used in Windows PowerShell 5.1. This is the preferred package manager for PowerShell. Both package managers have preregistered the PowerShell Gallery as a PSRepository. You can register other repositories, including private repositories that you host and maintain yourself.

You can create your own package repository as a local directory or network file share. You access file-share-based repositories using local filesystem or remote protocols, such as SMB or NFS. The following example shows how to register a local directory as a PSRepository.

```
PS> $registerPSRepositorySplat = @{
   Name = 'Local'
   Uri = 'D:/PSRepoLocal/'
   Trusted = $true
   Priority = 20
}
PS> Register-PSResourceRepository @registerPSRepositorySplat
PS> Get-PSResourceRepository Local

Name  Uri                     Trusted Priority IsAllowedByPolicy
----  ---                     ------- -------- -----------------
Local file:///D:/PSRepoLocal/ True    20       True
```

The PSResourceGet module also supports several other repository types:

- NuGet.org
- Azure Artifacts
- Azure Container Registry
- Microsoft Artifact Registry (MAR)
- GitHub Packages
- JFrog Artifactory
- MyGet.org
- Hosting your own NuGet.Server instance

Publish your module

The module manifest contains the metadata necessary to publish your module. It's also the best way to export the functions, variables, and aliases that you want to make public. Use the `New-ModuleManifest` cmdlet to create the manifest. The command has parameters to set most of the configuration options in the manifest. However, it's easier to create a minimal manifest and edit it as you develop your module.

For more information, see the *Required Metadata for Items Published to the PowerShell Gallery* section of Creating and publishing an item - PowerShell Gallery. This article talks about the requirements for PowerShell Gallery, but the requirements are the same for all PSRepositories.

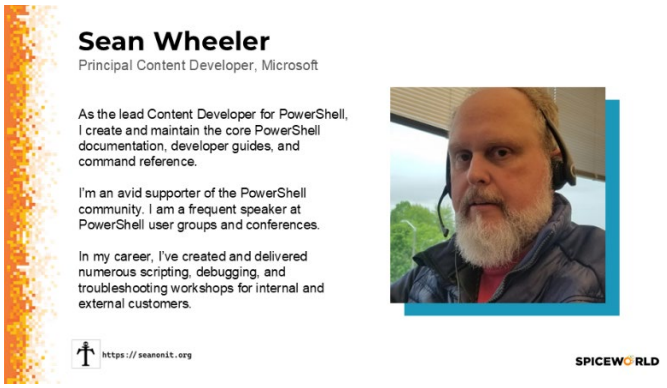
Once you have created your module manifest and completed coding, you need to assemble all the parts of the module into a single folder. The `Publish-PSResource` command reads the module manifest in the target folder, builds a NuGet package (`.nupkg`) file, and publishes it to the target Repository.

```
PS> Publish-PSResource -Path c:\MyModule -Repository Local
```

Further reading

- New-ModuleManifest
- Test-ModuleManifest
- Publish-PSResource

Presented by



Presentations:	https://seanonit.org/docs
GitHub:	https://github.com/sdwheeler
Bluesky:	@sdwheeler.bsky.social
LinkedIn:	in/scriptingsean

Presentations:	https://shortcutyour.life
GitHub:	https://github.com/stevenjudd
X/Twitter:	@stevenjudd
LinkedIn:	in/stevenjudd