Microsoft

Error handling in PowerShell

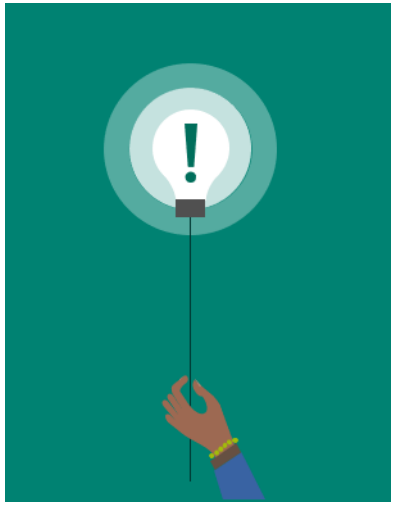
Module 6

Objectives

Learnings covered:

- Introduction to Error Handling
- Streams
- Write-Host and Streams
- Working with ErrorRecord Object
- Non-Terminating Errors
- Introduction to Terminating Errors
- Catching Terminating Errors using Trap
- Catching Terminating Errors Try {} Catch {} Finally {}

- Terminating and Non-Terminating Errors
- $Error variable
- $ErrorActionPreference variable
- BreakPoints



Introduction to Error Handling

What is an error?

PowerShell representation of an object-based exception

Design Time Errors

Syntax errors are caught by the PowerShell parser and/or PowerShell ISE

Runtime Errors

When something goes wrong during code execution

Only detectable when a specific state is reached, or statement is executed

Types of errors

Cmdlet/Function/Script/Object author decides which type of errors to trigger

Terminating

- Stops a statement from running
- If PowerShell does not handle the terminating error, it stops running the function or script

Non-Terminating

- Less severe, often just a single operation out of many
- Primary processing doesn't need to stop
- Error may be displayed in host application

What to do when an error occurs?

Nothing

- Red error messages are displayed
- Results could be unpredictable

Debug Code

- Identify where the error has occurred
- Resolve syntax or logic problems

Handle the Errors with Code

- Typically hide the red error messages
- Write logic to handle errors appropriately
- Actions can be: ignore, process, log, raise, or halt further execution

Error Handling: Terminating vs. Non-Terminating

Terminating Errors:

- Requires Try, Catch, Finally or a Trap statement
- Enters a child scope

Non-Terminating Errors:

- Test for error and run handling code
- Can be converted into terminating error
    - “Stop” with –ErrorAction or $ErrorActionPreference
    - Use the "throw" keyword

Questions?

Working with ErrorRecord Object

ErrorRecord Object

Is stored in and accessed via **$Error** array automatic variable

Is an instance of ErrorRecord Class in **System.Management.Automation** Namespace

Is an **extension** to **System.Exception** .Net class with PowerShell with additional PowerShell related properties

The actual System.Exception is stored in **Exception property** of the ErrorRecord Class

The PowerShell host investigates ErrorRecord Objects and determines to stop or not the pipeline

$Error – An array of ErrorRecord objects

- Automatic variable – Array holds errors that have occurred in the current session
- Control maximum array size:

```
PS C:\> $MaximumErrorCount
256
```

- The most recent error is the first object in the array:

```
PS C:\> $Error[0]
```

- Prevent an error from being added to the $Error array by using "-ErrorAction Ignore"

$Error – View all properties of ErrorRecord

- By default **$Error** only displays a summary of the error

- View all properties using one of the following examples:

```
PS C:\> $Error[0] | Format-List * -Force
PS C:\> $Error[0] | fl * -Force
PS C:\> $Error[0] | Select-Object *
PS C:\> $Error[0] | Select *
```

- Drill into the object model for more error details

```
PS C:\> $Error[0].CategoryInfo
PS C:\> $Error[0].InvocationInfo
PS C:\> $Error[0].ScriptStackTrace
```

$? – Last operation execution status

Automatic Variable

- Contains execution status of **last operation**
- Applies to both terminating and non-terminating errors
- Even applies to **external command** exit codes

- True = Complete Success
- False = Failure (Partial or Complete)

- Typically used in an If() statement to test and then run error handling code

-ErrorVariable Common Parameter

Capture errors from a specific action into a dedicated variable

Create the variable and store any errors in it:

```
Get-Process -Id 6 -ErrorVariable MyErrors
```

Append error messages to the variable:

```
Get-Process -Id 6 -ErrorVariable +MyErrors
```

Work with the variable just like $Error:

```
$MyErrors
```

$Error holds **all errors**, including those sent to **-ErrorVariable**.

Demonstration

Working with ErrorRecord Object

Questions?

Non-Terminating Errors

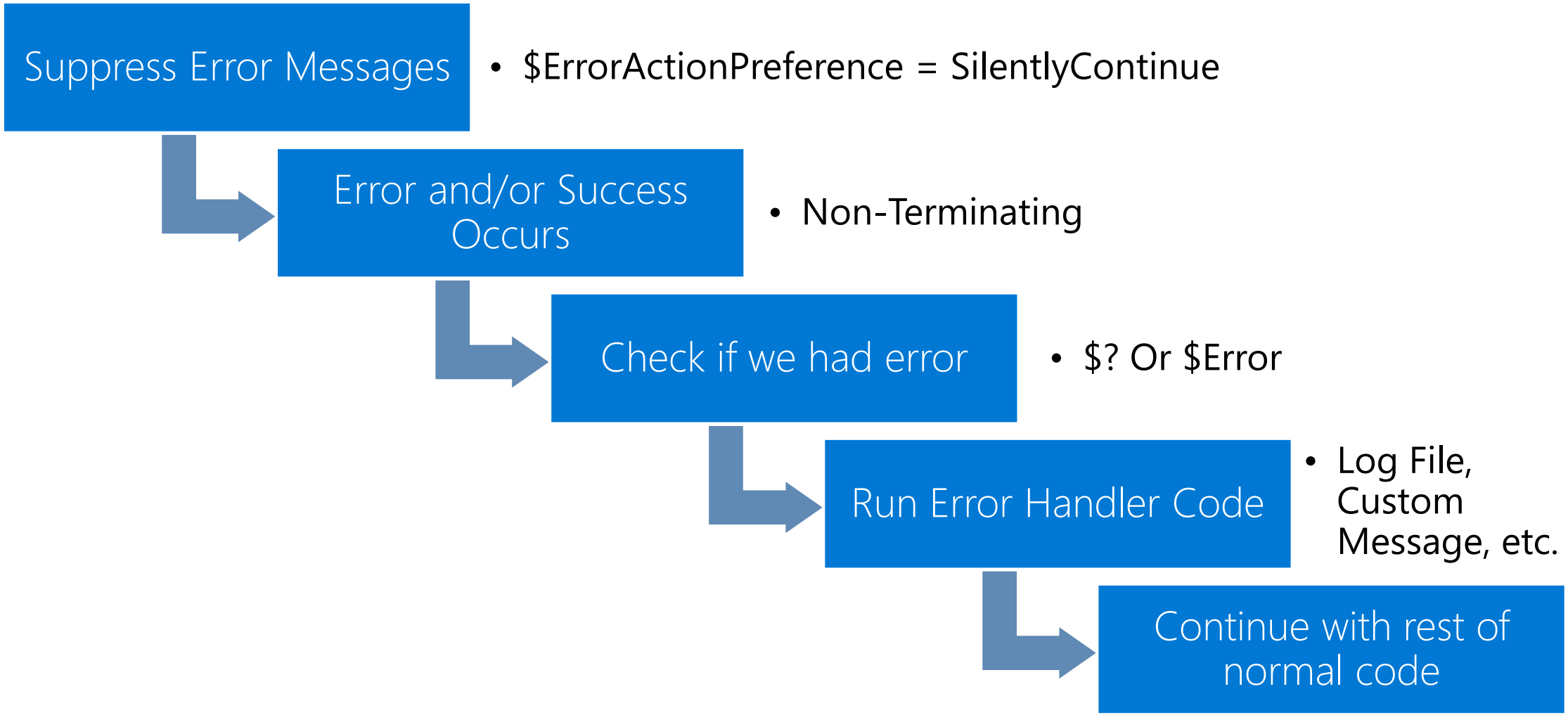
Errors do not stop processing

- Optionally handle error in code
- Continue with rest of code
- Does not trigger Trap, Try..Catch..Finally

Can be converted into Terminating errors

- -ErrorAction Stop or $ErrorActionPreference = ‘Stop’
- Can then utilize Trap, Try..Catch..Finally

Non-Terminating error handling flow



Actions to Suppress Error Messages

$ErrorActionPreference

- Automatic variable for remaining script/function/session
- Affects all the Cmdlets and Advanced Functions in Current scope

-ErrorAction common parameter

- On all cmdlets and advanced functions/scripts
- Affects only command where used
- Use SilentlyContinue value to suppress Error Messages

-ErrorAction Values

Determines PowerShell response to non-terminating errors

Optional Numerical Values	
SilentlyContinue	0
Stop	1
Continue	2
Inquire	3
Ignore	4
Suspend	5

Continue (Default)	• Displays error message and continues executing
SilentlyContinue	• Error message is not displayed, and execution continues w/o interruption
Stop	• Raises terminating error and displays an error message and stops command execution
Suspend	• Automatically suspends a workflow job • Allows for investigation, can be resumed
Inquire	• Displays error message and prompts user to continue
Ignore	• Can only be used with -ErrorAction common parameter (not with $ErrorActionPreference)

$ErrorActionPreference

Variable value affects all subsequent commands in same variable scope

- Errors are displayed by default:

```
PS C:\> $ErrorActionPreference
Continue
```

- Visible Errors Suppressed:

```
PS C:\>  $ErrorActionPreference = "SilentlyContinue"
 # or
PS C:\>  $ErrorActionPreference = 0
```

Write-Error

- Creates non-terminating errors
- Can be handled in-code via $?
- Automatically stored in $Error
- Useful for re-usable functions to report errors to caller

```
PS C:\> Write-Error "My Custom Error"
Write-Error "My Custom Error" : My Custom Error
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException

PS C:\> $Error[0]
Write-Error "My Custom Error" : My Custom Error
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
```

Demonstration

Non-Terminating Errors

Questions?

Introduction to Terminating Errors

Terminating Errors

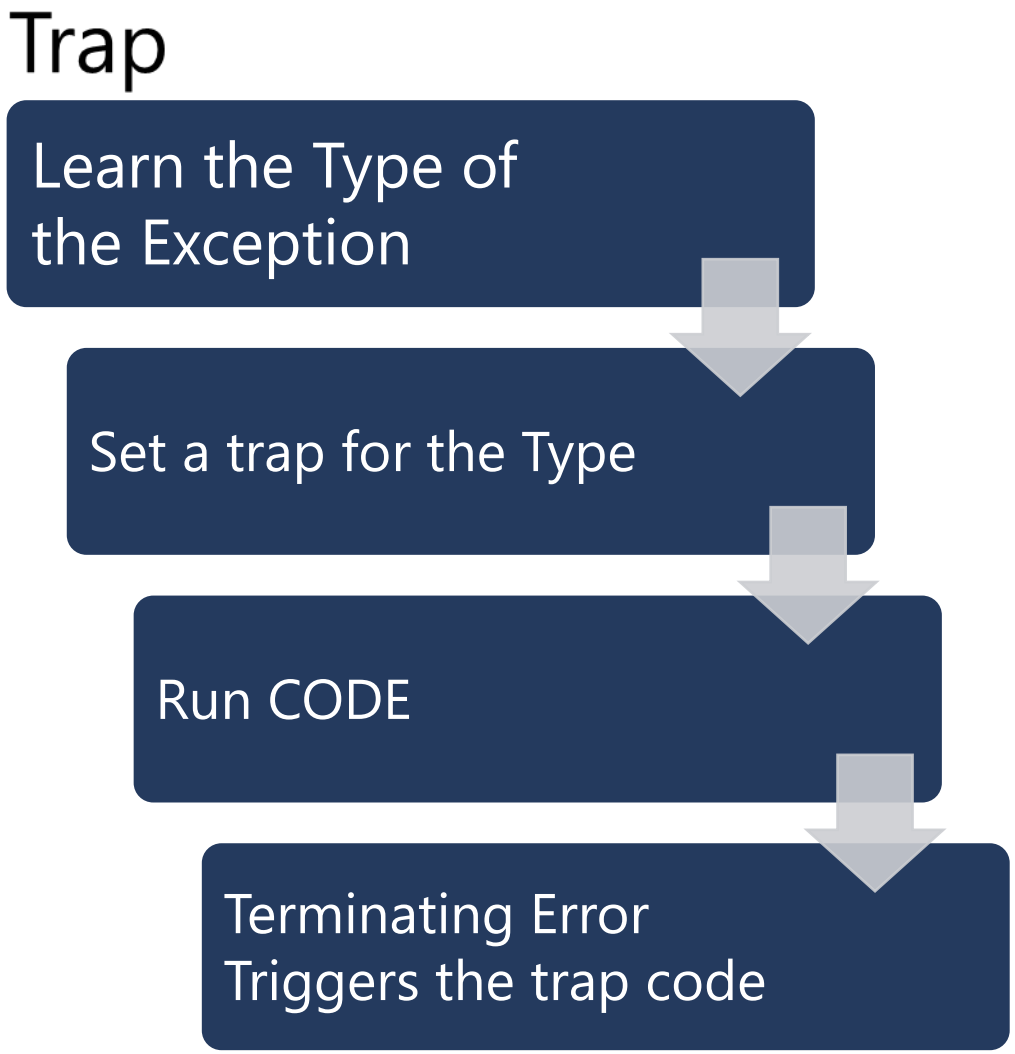
Severe errors where processing of the current scope can’t continue

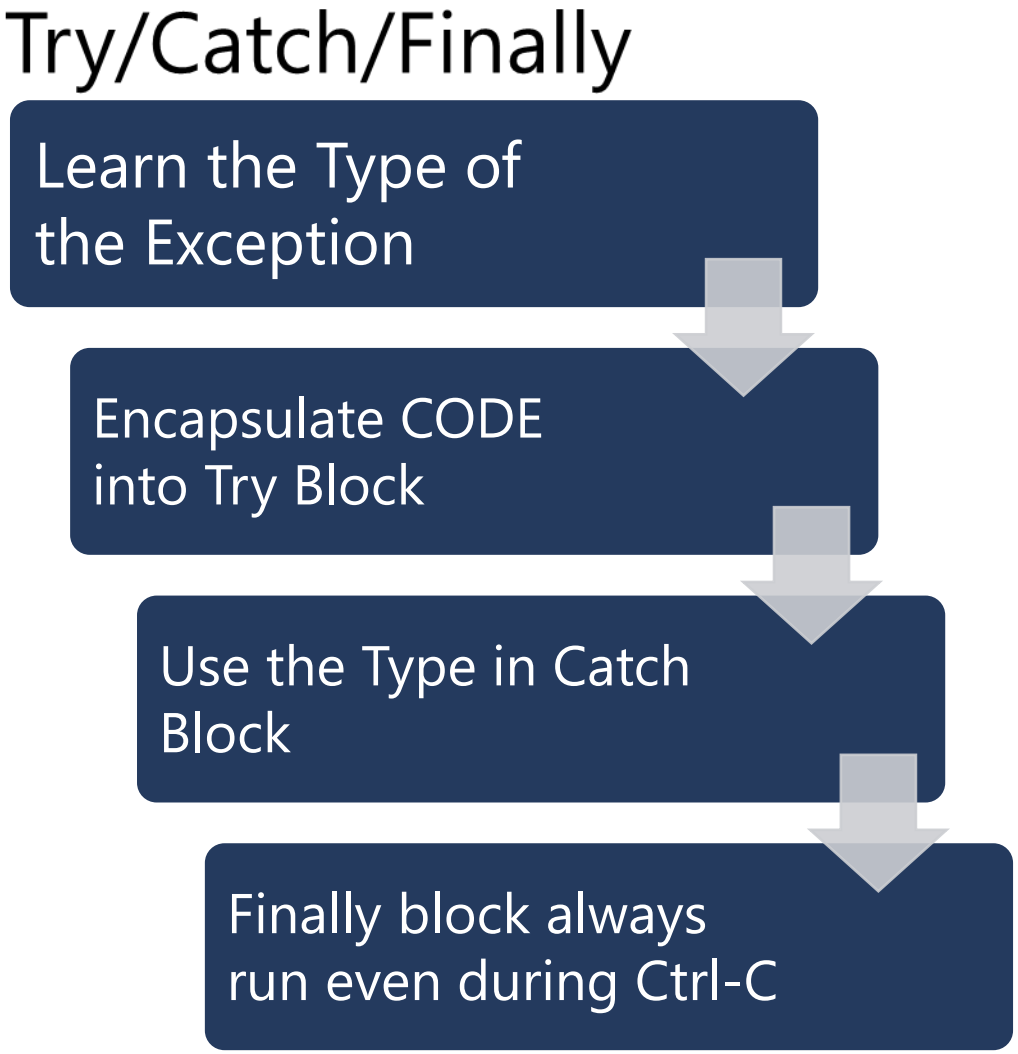
Error cannot be handled in the same code block, like non-terminating errors

Use Trap or Try{} Catch{} Finally{} to handle

Non-Terminating Errors can be re-thrown as terminating errors using "-ErrorAction stop"

Overview of Trap/Try..Catch..Finally





Type of the Exception

Both Trap and Try{} Catch{} Finally{} Catches the Type (Class) of the Exception Object

It can be specific or generic. Common Exception Types;

- SystemException
- IndexOutOfRangeException
- NullReferenceException
- InvalidOperationException
- ArgumentException
- ArgumentNullException
- Argument out of range exception

Specific

Common

To learn the current ErrorRecords ExceptionType, use FullName;

```
PS C:\>   $Error[0].Exception.InnerException.GetType() | Select-object -
Property Name, BaseType, FullName | fl

Name     : DivideByZeroException
BaseType : System.ArithmeticException
FullName : System.DivideByZeroException
```

Throw

- User-created terminating error.
- Throws a message string or any object type.
- Can be used to enforce mandatory parameters
  - PowerShell 3.0+, use the [Parameter(Mandatory)] attribute
- Useful in re-usable code to cause halt and report errors to caller for severe errors

```
PS C:\>  If ( -not (Test-Path $UserFile)) { Throw "ERROR: File not found" }
```

Example: Trace parameter binding

Basic Throw

```
If (1 -eq 1)
{
    "Line before the terminating error"
    Throw "This is my custom terminating error"
    "Line after the throw"
}
```

Line after throw doesn't run because of termination

```
PS C:\> C:\Simple-Throw-Sample.ps1
Line before the terminating error
This is my custom terminating error
At C:\Users\danpark\OneDrive @ Microsoft\Scripting\Simple-Throw-Sample.ps1:4
char:5
+     Throw "This is my custom terminating error"
+     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (This is my custom terminating
    error:String) [], RuntimeException
    + FullyQualifiedErrorId : This is my custom terminating error

PS C:\>
```

Demonstration

Terminating Errors

Questions?

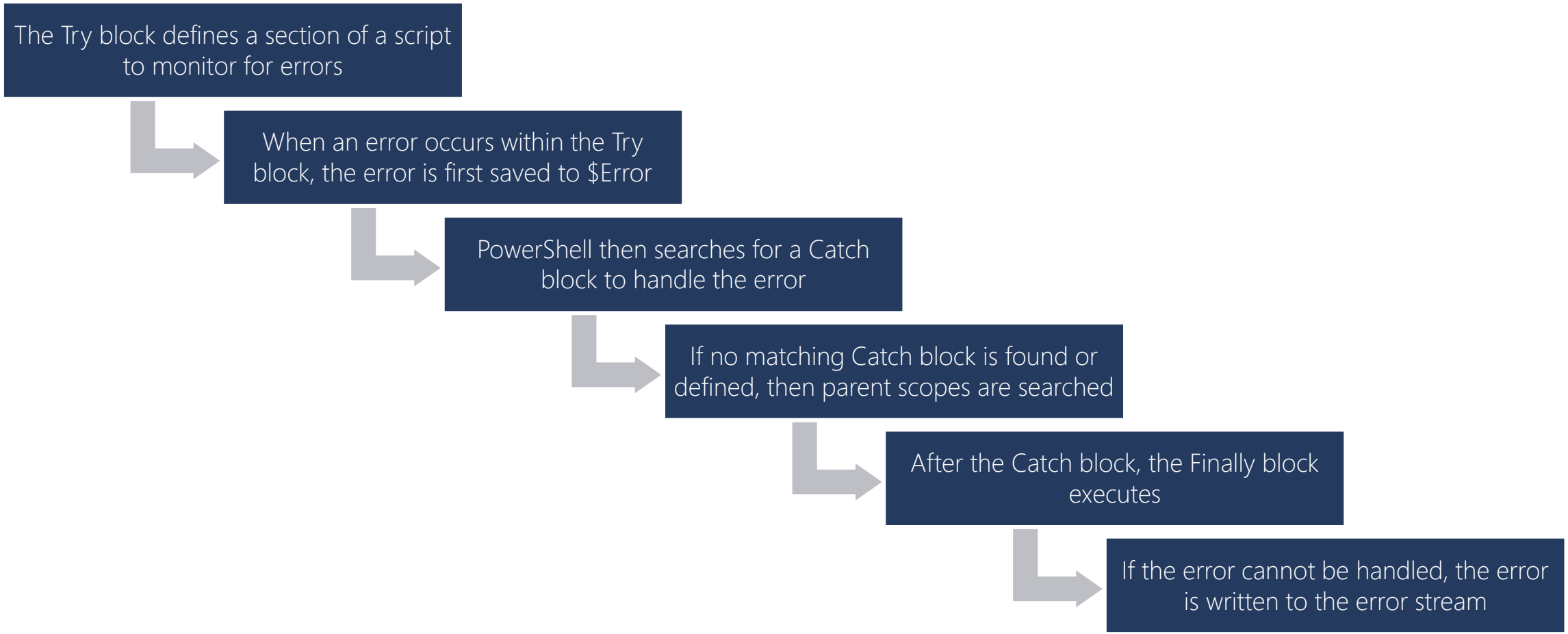
Catching Terminating Errors Try {} Catch {} Finally {}

Try, Catch, Finally Syntax

```
Try
{
    <Code that could cause terminating errors>
}

Catch [Exception Type1], [Exception Type2], [Exception Type3]
{
    <Error handling>
}

Catch
{
    <Handle uncaught errors>
}

Finally
{
    <Clean up code>
}
```

- One or more catch blocks
- Exception type optional
  Note: most specific to least specific
- Finally block optional
- Must have at least one Catch or Finally block

Try, Catch, Finally Flow



Finally Block

Will run even if Ctrl-C is used during the Try block execution

Useful for:

- Releasing resources
- Closing network connections
- Closing database connections
- Logging
- Etc.

Example: Try/Catch/Finally

```
function Get-TryCatchFinally {
    try {
        Get-Content -Path c:\notexist.txt -ErrorAction Stop
    }
    catch {
        Write-Output "Exception found: $($_.Exception.Message)"
    }
    finally {
        Write-Output 'Finally Block Code'
    }
}

Get-TryCatchFinally
```

$_ contains caught error

Scopes and Try, Catch, Finally

Exceptions can be re-thrown to the parent scope

- i.e. from a function to the calling scope (exception 'bubbling')
- Unhandled exceptions in any scope will be processed by the Windows PowerShell Host

In the code below

- The "function3" catch block is executed, which throws a new exception
- The new exception is caught by the parent catch block
- This affects flow control, as the "function3 was completed" text is NOT executed

```
$ErrorActionPreference = SilentlyContinue

Function function3 {
    Try {NonsenseCommand}
    Catch {"Error trapped inside function" ; Throw}
    "Function3 was completed"
}

Try {Function3}
Catch { "Internal Function error re-thrown: $($_.ScriptStackTrace)"}
"Script Completed"
```

Demonstration

Catching Terminating Errors
using Try{} Catch{} Finally{}

Questions?