**Index**

* Index
* Basic terminology
  + Exception
  + Throw and Catch
  + The call stack
  + Terminating and non-terminating errors
  + Swallowing an exception
* Basic command syntax
  + Throw
    - Write-Error -ErrorAction Stop
    - Cmdlet -ErrorAction Stop
  + Try/Catch
  + Try/Finally
  + Try/Catch/Finally
* $PSItem
  + PSItem.ToString()
  + $PSItem.InvocationInfo
  + $PSItem.ScriptStackTrace
  + $PSItem.Exception
    - $PSItem.Exception.Message
    - $PSItem.Exception.InnerException
    - $PSItem.Exception.StackTrace
* Working with exceptions
  + Catching typed exceptions
  + Catch multiple types at once
  + Throwing typed exceptions
    - Write-Error -Exception
    - The big list of .Net exceptions
  + Exceptions are objects
  + Re-throwing an exception
    - Re-throwing a new exception
    - $PSCmdlet.ThrowTerminatingError()
* Try can create terminating errors
  + $PSCmdlet.ThrowTerminatingError() inside try/catch
  + Public function templates
* Trap
* Closing remarks

# Basic terminology

We need to cover some basic terms before we jump into this one.

## Exception

An Exception is like an event that is created when normal error handling can not deal with the issue. Trying to divide a number by zero or running out of memory are examples of something that will create an exception. Sometimes the author of the code you are using will create exceptions for certain issues when they happen.

## Throw and Catch

When an exception happens, we say that an exception is thrown. To handle a thrown exception, you need to catch it. If an exception is thrown and it is not caught by something, the script will stop executing.

## The call stack

The call stack is the list of functions that have called each other. When a function is called, it gets added to the stack or the top of the list. When the function exits or returns, it will be removed from the stack.

When an exception is thrown, that call stack is checked in order for an exception handler to catch it.

## Terminating and non-terminating errors

An exception is generally a terminating error. A thrown exception will either be caught or it will terminate the current execution. By default, a non-terminating error is generated by Write-Error and it adds an error to the output stream without throwing an exception.

I point this out because Write-Error and other non-terminating errors will not trigger the catch.

## Swallowing an exception

This is when you catch an error just to suppress it. Do this with caution because it can make troubleshooting issues very difficult.

# Basic command syntax

Here is a quick overview of the basic exception handling syntax used in PowerShell.

## Throw

To create our own exception event, we throw an exception with the throw keyword.

function Do-Something

{

throw "Bad thing happened"

}

This creates a runtime exception that is a terminating error. It will be handled by a catchin a calling function or exit the script with a message like this.

PS:> Do-Something

Bad thing happened

At line:1 char:1

+ throw "Bad thing happened"

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : OperationStopped: (Bad thing happened:String) [], RuntimeException

+ FullyQualifiedErrorId : Bad thing happened

### Write-Error -ErrorAction Stop

I mentioned that Write-Error does not throw a terminating error by default. If you specify -ErrorAction Stop then Write-Errorgenerates a terminating error that can be handled with a catch.

Write-Error -Message "Houston, we have a problem." -ErrorAction Stop

Thank you to Lee Daily for reminding about using -ErrorAction Stop this way.

### Cmdlet -ErrorAction Stop

If you specify -ErrorAction Stop on any advanced function or Cmdlet, it will turn all Write-Error statements into terminating errors that will stop execution or that can be handled by a catch.

Do-Something -ErrorAction Stop

## Try/Catch

The way exception handling works in PowerShell (and many other languages) is that you first try a section of code and if it throws an error, you can catch it. Here is a quick sample.

try

{

Do-Something

}

catch

{

Write-Output "Something threw an exception"

}

try

{

Do-Something -ErrorAction Stop

}

catch

{

Write-Output "Something threw an exception or used Write-Error"

}

The catch script only runs if there is a terminating error. If the try executes correctly, then it will skip over the catch.

## Try/Finally

Sometimes you don’t need to handle an error but still need some code to execute if an exception happens or not. A finally script does exactly that.

Take a look at this example:

$command = [System.Data.SqlClient.SqlCommand]::New(queryString, connection)

$command.Connection.Open()

$command.ExecuteNonQuery()

$command.Connection.Close()

Any time you open or connect to a resource, you should close it. If the ExecuteNonQuery()throws an exception, the connection will not get closed. Here is the same code inside a try/finally block.

$command = [System.Data.SqlClient.SqlCommand]::New(queryString, connection)

try

{

$command.Connection.Open()

$command.ExecuteNonQuery()

}

finally

{

$command.Connection.Close()

}

In this example, the connection will get closed if there is an error. It will also get closed if there is no error. The finally script will run every time.

Because you are not catching the exception, it will still get propagated up the call stack.

## Try/Catch/Finally

It is perfectly valid to use catch and finally together. Most of the time you will use one or the other, but you may find scenarios where you will use both.

# $PSItem

Now that we got the basics out of the way, we can dig a little deeper.

Inside the catch block, there is an automatic variable ($PSItem or $\_) of type ErrorRecord that contains the details about the exception. Here is a quick overview of some of the key properties.

For these examples, I used an invalid path in ReadAllText to generate this exception.

[System.IO.File]::ReadAllText( '\\test\no\filefound.log')

## PSItem.ToString()

This will give you the cleanest message to use in logging and general output. ToString()is automatically called if $PSItem is placed inside a string.

catch

{

Write-Output "Ran into an issue: $($PSItem.ToString())"

}

catch

{

Write-Output "Ran into an issue: $PSItem"

}

## $PSItem.InvocationInfo

This property contains additional information collected by PowerShell about the function or script where the exception was thrown. Here is the InvocationInfo from the sample exception that I created.

PS:> $PSItem.InvocationInfo | Format-List \*

MyCommand : Get-Resource

BoundParameters : {}

UnboundArguments : {}

ScriptLineNumber : 5

OffsetInLine : 5

ScriptName : C:\blog\throwerror.ps1

Line : Get-Resource

PositionMessage : At C:\blog\throwerror.ps1:5 char:5

+ Get-Resource

+ ~~~~~~~~~~~~

PSScriptRoot : C:\blog

PSCommandPath : C:\blog\throwerror.ps1

InvocationName : Get-Resource

The important details here show the ScriptName, the Line of code and the ScriptLineNumber where the invocation started.

## $PSItem.ScriptStackTrace

This property will show the order of function calls that got you to the code where the exception was generated.

PS:> $PSItem.ScriptStackTrace

at Get-Resource, C:\blog\throwerror.ps1: line 13

at Do-Something, C:\blog\throwerror.ps1: line 5

at <ScriptBlock>, C:\blog\throwerror.ps1: line 18

I am only making calls to functions in the same script but this would track the calls if multiple scripts were involved.

## $PSItem.Exception

This is the actual exception that was thrown.

### $PSItem.Exception.Message

This is the general message that describes the exception and is a good starting point when troubleshooting. Most exceptions have a default message but can also be set to something custom when the exception is thrown.

PS:> $PSItem.Exception.Message

Exception calling "ReadAllText" with "1" argument(s): "The network path was not found."

This is also the message returned when calling $PSItem.ToString() if there was not one set on the ErrorRecord.

### $PSItem.Exception.InnerException

Exceptions can contain inner exceptions. This is often the case when the code you are calling catches an exception and throws a different exception. They will place the original exception inside the new exception.

PS:> $PSItem.Exception.InnerExceptionMessage

The network path was not found.

I will revisit this later when I talk about re-throwing exceptions.

### $PSItem.Exception.StackTrace

This is the StackTrace for the exception. I showed a ScriptStackTrace above, but this one is for the calls to managed code.

at System.IO.\_\_Error.WinIOError(Int32 errorCode, String maybeFullPath)

at System.IO.FileStream.Init(String path, FileMode mode, FileAccess access, Int32 rights, Boolean useRights, FileShare share, Int32 bufferSize, FileOptions options, SECURITY\_ATTRIBUTES secAttrs, String msgPath, Boolean bFromProxy, Boolean useLongPath, Boolean checkHost)

at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess access, FileShare share, Int32 bufferSize, FileOptions options, String msgPath, Boolean bFromProxy, Boolean useLongPath, Boolean checkHost)

at System.IO.StreamReader..ctor(String path, Encoding encoding, Boolean detectEncodingFromByteOrderMarks, Int32 bufferSize, Boolean checkHost)

at System.IO.File.InternalReadAllText(String path, Encoding encoding, Boolean checkHost)

at CallSite.Target(Closure , CallSite , Type , String )

You will only get this stack trace when the event is thrown from managed code. I am calling a .Net framework function directly so that is all we can see in this example. Generally when you are looking at a stack trace, you are looking for where your code stops and the system calls begin.

# Working with exceptions

There is more to exceptions than the basic syntax and exception properties.

## Catching typed exceptions

You can be selective with the exceptions that you catch. Exceptions have a type and you can specify the type of exception you want to catch.

try

{

Do-Something -Path $path

}

catch [System.IO.FileNotFoundException]

{

Write-Output "Could not find $path"

}

catch [System.IO.IOException]

{

Write-Output "IO error with the file: $path"

}

The exception type is checked for each catch block until one is found that matches your exception. It is important to realize that exceptions can inherit from other exceptions. In the example above, FileNotFoundException inherits from IOException. So if the IOException was first, then it would get called instead. Only one catch block will be invoked even if there are multiple matches.

If we had a System.IO.PathTooLongException then the IOException would match but if we had a InsufficientMemoryException then nothing would catch it and it would propagate up the stack.

## Catch multiple types at once

It is possible to catch multiple exception types with the same catch statement.

try

{

Do-Something -Path $path -ErrorAction Stop

}

catch [System.IO.DirectoryNotFoundException],[System.IO.FileNotFoundException]

{

Write-Output "The path or file was not found: [$path]"

}

catch [System.IO.IOException]

{

Write-Output "IO error with the file: [$path]"

}

Thank you /u/Sheppard\_Ra for suggesting this addition.

## Throwing typed exceptions

You can throw typed exceptions in PowerShell. Instead of calling throw with a string:

throw "Could not find: $path"

Use an exception accelerator like this:

throw [System.IO.FileNotFoundException] "Could not find: $path"

But you have to specify a message when you do it that way.

You can also create a new instance of an exception to be thrown. The message is optional when you do this because the system has default messages for all built in exceptions.

throw [System.IO.FileNotFoundException]::new()

throw [System.IO.FileNotFoundException]::new("Could not find path: $path")

If you are not yet using PowerShell 5.0, you will have to use the older New-Objectapproach.

throw (New-Object -TypeName System.IO.FileNotFoundException )

throw (New-Object -TypeName System.IO.FileNotFoundException -ArgumentList "Could not find path: $path")

By using a typed exception, you (or others) can catch the exception by the type as mentioned in the previous section.

### Write-Error -Exception

We can add these typed exceptions to Write-Error and we can still catch the errors by exception type. Use Write-Error like in these examples:

# with normal message

Write-Error -Message "Could not find path: $path" -Exception ([System.IO.FileNotFoundException]::new()) -ErrorAction Stop

# With message inside new exception

Write-Error -Exception ([System.IO.FileNotFoundException]::new("Could not find path: $path")) -ErrorAction Stop

# Pre PS 5.0

Write-Error -Exception ([System.IO.FileNotFoundException]"Could not find path: $path") -ErrorAction Stop

Write-Error -Message "Could not find path: $path" -Exception ( New-Object -TypeName System.IO.FileNotFoundException ) -ErrorAction Stop

Then we can catch it like this:

catch [System.IO.FileNotFoundException]

{

Write-Log $PSItem.ToString()

}

### The big list of .Net exceptions

I compiled a master list with the help of the Reddit/r/PowerShell community that contains hundreds of .Net exceptions to complement this post.

* The big list of .Net exceptions

I start by searching that list for exceptions that feel like they would be a good fit for my situation. You should try to use exceptions in the base System namespace.

## Exceptions are objects

If you start using a lot of typed exceptions, remember that they are objects. Different exceptions have different constructors and properties. If we look at the documentationfor System.IO.FileNotFoundException, we will see that we can pass in a message and a file path.

[System.IO.FileNotFoundException]::new("Could not find file", $path)

And it has a FileName property that exposes that file path.

catch [System.IO.FileNotFoundException]

{

Write-Output $PSItem.Exception.FileName

}

You will have to consult the .Net documentation for other constructors and object properties.

## Re-throwing an exception

If all you are going to do in your catch block is throw the same exception, then don’t catch it. You should only catch an exception that you plan to handle or perform some action when it happens.

There are times where you want to perform an action on an exception but re-throw the exception so something downstream can deal with it. We could write a message or log the problem close to where we discover it but handle the issue further up the stack.

catch

{

Write-Log $PSItem.ToString()

throw $PSItem

}

Interestingly enough, we can call throw from within the catch and it will re-throw the current exception.

catch

{

Write-Log $PSItem.ToString()

throw

}

We want to re-throw the exception to preserve the original execution information like source script and line number. If we throw a new exception at this point it will hide where the exception started.

### Re-throwing a new exception

If you catch an exception but you want to throw a different one, then you should nest the original exception inside the new one. This allows someone down the stack to access it as the $PSItem.Exception.InnerException.

catch

{

throw [System.MissingFieldException]::new('Could not access field',$PSItem.Exception)

}

### $PSCmdlet.ThrowTerminatingError()

The one thing that I don’t like about using throw for raw exceptions is that the error message points at the throw statement and indicates that line is where the problem is.

Unable to find the specified file.

At line:31 char:9

+ throw [System.IO.FileNotFoundException]::new()

+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

+ CategoryInfo : OperationStopped: (:) [], FileNotFoundException

+ FullyQualifiedErrorId : Unable to find the specified file.

Having the error message tell me that my script is broken because I called throw on line 31 is a bad message for users of your script to see. It does not tell them anything useful.

Dexter Dhami pointed out that I can use ThrowTerminatingError() to correct that.

$PSCmdlet.ThrowTerminatingError(

[System.Management.Automation.ErrorRecord]::new(

([System.IO.FileNotFoundException]"Could not find $Path"),

'My.ID',

[System.Management.Automation.ErrorCategory]::OpenError,

$MyObject

)

)

If we assume that ThrowTerminatingError() was called inside a function called Get-Resource, then this is the error that we would see.

Get-Resource : Could not find C:\Program Files (x86)\Reference

Assemblies\Microsoft\Framework\.NETPortable\v4.6\System.IO.xml

At line:6 char:5

+ Get-Resource -Path $Path

+ ~~~~~~~~~~~~

+ CategoryInfo : OpenError: (:) [Get-Resource], FileNotFoundException

+ FullyQualifiedErrorId : My.ID,Get-Resource

Do you see how it points to the Get-Resource function as the source of the problem? That tells the user something useful.

Because $PSItem is an ErrorRecord, we can also use ThrowTerminatingError this way to re-throw.

catch

{

$PSCmdlet.ThrowTerminatingError($PSItem)

}

This will change the source of the error to the Cmdlet and hide the internals of your function from the users of your Cmdlet.

# Try can create terminating errors

Kirk Munro points out that some exceptions are only terminating errors when executed inside a try/catch block. Here is the example he gave me that generates a divide by zero runtime exception.

function Do-Something { 1/(1-1) }

Then invoke it like this to see it generate the error and still output the message.

&{ Do-Something; Write-Output "We did it. Send Email" }

But by placing that same code inside a try/catch, we see something else happen.

try

{

&{ Do-Something; Write-Output "We did it. Send Email" }

}

catch

{

Write-Output "Notify Admin to fix error and send email"

}

We see the error become a terminating error and not output the first message. What I don’t like about this one is that you can have this code in a function and it will act differently if someone is using a try/catch.

I have not ran into issues with this myself but it is corner case to be aware of.

## $PSCmdlet.ThrowTerminatingError() inside try/catch

One nuance of $PSCmdlet.ThrowTerminatingError() is that it creates a terminating error within your Cmdlet but it turns into a non-terminating error after it leaves your Cmdlet. This leaves the burden on the caller of your function to decide how to handle the error. They can turn it back into a terminating error by using -ErrorAction Stop or calling it from within a try{...}catch{...}.

## Public function templates

One last take a way I had with my conversation with Kirk Munro was that he places a try{...}catch{...} around every begin, process and end block in all of his advanced functions. In those generic catch blocks, he as a single line using $PSCmdlet.ThrowTerminatingError($PSitem) to deal with all exceptions leaving his functions.

function Do-Something

{

[cmdletbinding()]

param()

process

{

try

{

...

}

catch

{

$PSCmdlet.ThrowTerminatingError($PSitem)

}

}

}

Because everything is in a try statement within his functions, everything acts consistently. This also gives clean errors to the end user that hides the internal code from the generated error.

# Trap

I focused on the try/catch aspect of exceptions. But there is one legacy feature I need to mention before we wrap this up.

A trap is placed in a script or function to catch all exceptions that happen in that scope. When an exception happens, the code in the trap will get executed and then the normal code will continue. If multiple exceptions happen, then the trap will get called over and over.

trap

{

Write-Log $PSItem.ToString()

}

throw [System.Exception]::new('first')

throw [System.Exception]::new('second')

throw [System.Exception]::new('third')

I personally never adopted this approach but I can see the value in admin or controller scripts that will log any and all exceptions, then still continues to execute.

# Closing remarks

Adding proper exception handling to your scripts will not only make them more stable, but it will also make it easier for you to troubleshoot those exceptions.

I spent a lot of time talking throw because it is a core concept when talking about exception handling. PowerShell also gave us Write-Error that handles all the situations where you would use throw. So don’t think that you need to be using throw after reading this.

Now that I have taken the time to write about exception handling in this detail, I am going to switch over to using Write-Error -Stop to generate errors in my code. I am also going to take Kirk’s advice and make ThrowTerminatingError my goto exception handler for every funciton.

Example:

# PowerShell Tutorial – Try Catch Finally and error handling in PowerShell

One of the key parts of any good PowerShell script is error handling. Even in the shortest script, being able to handle errors helps to ensure that an unexpected event will not go on to wreck the system you are working on. Take the example below. Every week in our sample company (MyCompany.Com) Human Resources are going to upload a list telling us who should have access to the Expenses database. If a name isn’t in the list from HR we’re going to remove it from the group and that user will no longer be able to log expense claims:

$AuthorizedUsers= Get-Content \\ FileServer\HRShare\UserList.txt

$CurrentUsers=Get-ADGroupMember "Expenses Claimants"

Foreach($User in $CurrentUsers)

{

If($AuthorizedUsers -notcontains $User)

{

Remove-ADGroupMember -Identity "Expenses Claimants" -User $User

}

}

Now, you can see where this is going to go wrong. One week HR doesn’t get around to uploading the list or, just as we are about to access the list, the file server dies. Suddenly PowerShell throws an error on the Get-Content cmdlet and the $AuthorizedUser variable remains empty. Because our script doesn’t handle errors, it continues to run and, in a very short space of time, it has removed every user from our expenses group. Pretty soon the irate phone calls start flooding in and life gets a little less happy. The way to avoid all this is to catch the errors and then handle the event that caused them (which in this case is halt the script and have a shout at someone in HR).

## Terminating and Non-Terminating Errors

One of the key things to know when catching errors is that only certain errors can be caught by default. Errors come in two types – terminating and non-terminating. A terminating error is an error that will halt a function or operation. If you make a syntax error or run out of memory, that is a terminating error. Terminating errors can be caught and handled. Non-terminating errors allow Powershell to continue and usually come from cmdlets or other managed situations. Under normal circumstances they cannot be caught by Try-Catch-Finally. The Get-Content error in the example above is a non-terminating error.

## Treating Non-Terminating Errors as Terminating

So how do you catch a Non-Terminating error? Basically, you tell PowerShell to treat it as terminating. To do this you use the ErrorAction parameter. Every PowerShell cmdlet supports ErrorAction. By specifying -ErrorAction Stop on the end of a cmdlet you ensure that any errors it throws are treated as terminating and can be caught. In our example above we are going to change our Get-Content line to:

$AuthorizedUsers= Get-Content \\ FileServer\HRShare\UserList.txt -ErrorAction Stop

## Treating All Errors as Terminating

It is also possible to treat all errors as terminating using the ErrorActionPreference variable. You can do this either for the script your are working with or for the whole PowerShell session. To set it in a script, make the first line $ErrorActionPreference = Stop. To set it for the session, type $ErrorActionPreference = Stop at the PowerShell console.

## Catching a Terminating Error

Once you have ensured that the error you are trying to catch is going to be treated as terminating, you can build a Try Catch block around the command (or commands) that might cause the error. The first stage is to surround the section of your script that may throw the error with a Try block. In our example the Get-Content line becomes:

Try

{

$AuthorizedUsers= Get-Content \\ FileServer\HRShare\UserList.txt -ErrorAction Stop

}

Immediately after the Try block you must place a Catch block to deal with the error. The Catch block is only accessed if a terminating error occurs, otherwise it is ignored. In our example we are going to email an admin to say that there has been an error and then halt the script. Our Get-Content line is now:

Try

{

$AuthorizedUsers= Get-Content \\ FileServer\HRShare\UserList.txt -ErrorAction Stop

}

Catch

{

Send-MailMessage -From ExpensesBot@MyCompany.Com -To WinAdmin@MyCompany.Com -Subject "HR File Read Failed!" -SmtpServer EXCH01.AD.MyCompany.Com

Break

}

## Accessing The Error Record

Once you are inside a catch block you can access the error record, which is stored in the current object variable, $\_. Error records have various useful properties, but the main one you will want to access is $\_.Exception. Exceptions are what we are really dealing with here as we catch and deal with errors – exceptions are the unexpected event that caused the error (the error record itself is actually only really a wrapper for presenting the exception to the PowerShell user). It is the exception that we are catching and the exception that contains all the really useful information about the problem. If there was a further underlying problem that caused our exception, it is also recorded at $\_.exception.innerexception (and so on – the next underlying exception is stored at $\_.exception.innerexception.innerexception etc.). For the purposes of our example we are going to use $\_.Exception to put some extra information into our notification email, using the $\_.Exception.Message and $\_.Exception.ItemName properties:

Try

{

$AuthorizedUsers= Get-Content \\ FileServer\HRShare\UserList.txt -ErrorAction Stop

}

Catch

{

$ErrorMessage = $\_.Exception.Message

$FailedItem = $\_.Exception.ItemName

Send-MailMessage -From ExpensesBot@MyCompany.Com -To WinAdmin@MyCompany.Com -Subject "HR File Read Failed!" -SmtpServer EXCH01.AD.MyCompany.Com -Body "We failed to read file $FailedItem. The error message was $ErrorMessage"

Break

}

## Catching Specific Exceptions

Now, as our example stands we are catching any errors that occur during the file read and dealing with all of them in the same way. You can however catch specific exceptions and deal with them differently, but – and it’s a big but – only if the original error is terminating. Because the Get-Content cmdlet throws non-terminating errors (that we have only treated as terminating using ErrorAction) we cannot specifically catch the different exceptions that the cmdlet might throw. This is a feature of PowerShell and applies to any non-terminating error, regardless of the ErrorActionPreference and cannot be changed. Still, we can deal with other terminating exceptions, such as an out of memory error, that could crop up during the read operation. For the purposes of this example that is what we will do.

You catch specific terminating errors by specifying the exception name immediately after the Catch keyword. In our example we want to catch a System.OutOfMemory exception and, if we get one, will take the no nonsense approach of rebooting the computer immediately. We will also include a general catch block after our file not found block to catch all other exceptions:

Try

{

$AuthorizedUsers= Get-Content \\ FileServer\HRShare\UserList.txt -ErrorAction Stop

}

Catch [System.OutOfMemoryException]

{

Restart-Computer localhost

}

Catch

{

$ErrorMessage = $\_.Exception.Message

$FailedItem = $\_.Exception.ItemName

Send-MailMessage -From ExpensesBot@MyCompany.Com -To WinAdmin@MyCompany.Com -Subject "HR File Read Failed!" -SmtpServer EXCH01.AD.MyCompany.Com -Body "We failed to read file $FailedItem. The error message was $ErrorMessage"

Break

}

## Finally, Using Finally

The last part of Try Catch Finally is the Finally block. This must be defined immediately after the Catch block and runs every time, regardless of whether there was an error or not. In this way you can perform actions that need to be made regardless of whether an operation succeeds or fails. In our example we are going to log that a file read was attempted. Our Get-Content line now looks like:

Try

{

$AuthorizedUsers = Get-Content \\ FileServer\HRShare\UserList.txt -ErrorAction Stop

}

Catch [System.OutOfMemoryException]

{

Restart-Computer localhost

}

Catch

{

$ErrorMessage = $\_.Exception.Message

$FailedItem = $\_.Exception.ItemName

Send-MailMessage -From ExpensesBot@MyCompany.Com -To WinAdmin@MyCompany.Com -Subject "HR File Read Failed!" -SmtpServer EXCH01.AD.MyCompany.Com -Body "We failed to read file $FailedItem. The error message was $ErrorMessage"

Break

}

Finally

{

$Time=Get-Date

"This script made a read attempt at $Time" | out-file c:\logs\ExpensesScript.log -append

}