Name Click here to enter name ID Click here to enter id

Errors and Exception Handling

Exercise 2.10

# Instructions

Answer all questions directly in this document. You will save and upload this completed document as your homework submission.

# Overview

Understanding Error handling is a key to resilient scripting and code. You will explore nonterminating and terminating errors in these exercises.

Nonterminating errors are errors that will not cause a command to quit executing. Nonterminating errors can be handled with the built-in common parameters ErrorAction and ErrorVariable. By default, the errors are output to the error stream and added to the $error collection.

Terminating errors are errors that will stop a command or expression. The preference variable **$ErrorActionPreference** determines what happens when a terminating error happens while a script is running. By default, **$ErrorActionPreference** is set to **Continue**, which means that the script *won’t* terminate, but will display the terminating error message and continue executing. If you change **$ErrorActionPreference** to **Stop**, then the error that terminated the cmdlet will also terminate the script.

Terminating errors are handled with try/catch statements. You will write a couple of try/catch blocks of code to get familiar with them.

## Requirements

* PowerShell
* Visual Studio Code

# Setup

* Optional: change to the temporary folder, **C:\Temp** (windows) or **/tmp** (Linux).
* Create some files to work with. Enter the command:   
  1..10|ForEach-Object{New-Item -Type File -Name "File$($\_\*2).demo"}   
  This will create the files: File2.demo, File4.demo, File6.demo, File8.demo, File10.demo, File12.demo, File14.demo, File16.demo, File18.demo, File20.demo in the current directory.
* Verify with the command:  
  dir \*.demo

# Task 1—$Error

Explore nonterminating errors and the **$Error** collection. Whenever an error occurs in PowerShell, PowerShell adds the error to the **$Error** collection. **$Error** has a max size of **$MaximumErrorCount**.

## Steps

1. Start a new PowerShell session.
2. The count property holds the number of errors in the $Error collection:  
   $Error.Count
   1. How many errors are in **$Error**? Click or tap here to enter text.
3. Delete the file **file2.demo**:  
   Remove-Item file2.demo
   1. What is the count of the **$error** collection? Click or tap here to enter text.
4. Try to delete the file again, this should cause an error.  
   Remove-Item file2.demo
   1. What is the count of the **$error** collection? Click or tap here to enter text.
5. Delete the file file3.demo  
   Remove-Item file3.demo
   1. What is the count of the **$error** collection? Click or tap here to enter text.
6. View the **$error** collection:  
   $Error
   1. Which error is listed first? Click or tap here to enter text.
7. As you can see the latest error is the first one contained in the list. You can access the most recent error using the 0 index. Enter:  
   $Error[0]
   1. Which error is displayed? Click or tap here to enter text.

# Task 2—ErrorAction and Error Variable

The common parameters **ErrorAction** and **ErrorVariable** can be used to control PowerShell’s nonterminating error behavior.

## Steps

1. Use **ErrorAction** to change the behavior of errors for a specific cmdlet. Enter the following command (remember there is no **file5.demo**):  
   Remove-Item file5.demo -ErrorAction SilentlyContinue
   1. What was the result? Click or tap here to enter text.
   2. Was the error placed in the **$Error** collection? Click or tap here to enter text.
2. Use an error variable to capture the error from a specific command. Enter (all on one line):  
   Remove-Item file7.demo -ErrorAction SilentlyContinue -ErrorVariable DelError
   1. What was returned? Click or tap here to enter text.
   2. View the contents of **$DelError**:  
      **$DelError**
      1. What does it contain? Click or tap here to enter text.
   3. Does **$Error** contain the error also? Click or tap here to enter text.
3. Now delete **file4.demo**. This should be successful:  
   del file4.demo -ErrorAction SilentlyContinue -ErrorVariable DelError
   1. What does **$DelError** contain? Click or tap here to enter text.
   2. What is the count of **$DelError**? Click or tap here to enter text.
   3. What does that tell you about the **ErrorVariable** when there are no errors? Click or tap here to enter text.
4. The **$ErrorActionPreference** variable contains the default value for **ErrorAction**. If you don’t specify an **ErrorAction** for a cmdlet the value in **$ErrorActionPreference** will be used.
   1. What is the value of **$ErrorActionPreference**? Click or tap here to enter text.

# Task 3—The Error Object

When an error is encountered the text of the error is written to the error stream, but an error object is stored in the **$Error** collection.

## Steps

1. Exploring the **Error** Object
   1. Get the error object type.   
      $Error[0].gettype()
   2. What object type is stored in the **$Error** collection?
   3. View the members of the object:  
      $error[0]|Get-Member
   4. For details see <https://docs.microsoft.com/en-us/powershell/scripting/developer/cmdlet/interpreting-errorrecord-objects>

# Task 4—Multiple Errors

Multiple non-terminating errors may occur during the execution of a cmdlet. Let’s see what happens.

## Steps

1. Let’s try to delete some files using a foreach loop.
   1. What is the count of $Error? Click or tap here to enter text.
   2. Use this command to delete the files **file10.demo** through **file16.demo**. Enter (all on one line):   
      **10..16 | ForEach-Object {Remove-Item "File$\_.demo" -ErrorAction Continue -ErrorVariable delerror}**
   3. Did you see more than one error message? Click or tap here to enter text.
   4. What is the count of $Error? Click or tap here to enter text.
   5. How many new errors were added? Click or tap here to enter text.
   6. What is in $DelError? Click or tap here to enter text.
   7. Explain? Click or tap here to enter text.
2. Explore multiple errors from one cmdlet. To get ready for this, you need to mark a couple of our remaining files as read only. To do that you will set the **ItemProperty**. You should have 4 demo files left. Verify with  
    **dir \*.demo**
   1. You should see something similar to this:   
        
      Mode LastWriteTime Length Name  
      ---- ------------- ------ ----  
      -a---- 5/11/2020 10:39 AM 0 file18.demo  
      -a---- 5/11/2020 10:39 AM 0 file20.demo  
      -a---- 5/11/2020 10:16 AM 0 File6.demo  
      -a---- 5/11/2020 10:16 AM 0 File8.demo
   2. Mark files 6 and 8 as readonly:   
      Set-ItemProperty -Path file6.demo -Name IsReadOnly -Value $true  
      Set-ItemProperty -Path file8.demo -Name IsReadOnly -Value $true
   3. Verify with:   
       **dir \*.demo**   
      Your results should look like this:

Mode LastWriteTime Length Name  
---- ------------- ------ ----  
-a---- 5/11/2020 10:39 AM 0 file18.demo  
-a---- 5/11/2020 10:39 AM 0 file20.demo  
-ar--- 5/11/2020 10:16 AM 0 File6.demo  
-ar--- 5/11/2020 10:16 AM 0 File8.demo   
  
Notice the extra **r** in the Mode property; those two files are now marked readonly.

* 1. Now try to delete all the demo files:  
     **del \*.demo -ErrorVariable delerror -Verbose**
  2. How many items were added to **$Error**? Click or tap here to enter text.
  3. What is in **$DelError**? Click or tap here to enter text.

1. To clean up, get rid of the remaining demo files. To delete files marked as readonly, use the **-force** option. Enter:   
   del \*.demo -force

# Task 5—Try/Catch

Terminating errors are errors that will stop a command or expression. Launch VS Code to write a script.

## Steps

1. Create a script file named **term.ps1** .
2. Write and save the following code:

$dividend=Read-Host "Enter a dividend: "

$divisor=Read-Host "Enter a divisor: "

$quotient=$Dividend/$divisor

Write-Host "The answer my friend is $quotient" -f green

* 1. Run the script entering **10** for the dividend and **2** for the divisor. What is the output?   
      Click or tap here to enter text.
  2. Run the script again, entering **10** for the dividend and **0** for the divisor. What was the output   
      Click or tap here to enter text.
  3. Run the script again, entering **10** for the dividend and **foo** for the divisor. What was the output?  
      Click or tap here to enter text.

1. What’s the default setting of your error action preference variable?  
   **$ErrorActionPreference**   
   Your output: Click or tap here to enter text.
   1. In your VSCode terminal pane, change the error action preference variable to **Stop**:  
      **$ErrorActionPreference = "Stop"**
   2. Run the script again, entering **10** for the dividend and **0** for the divisor. Then run the script again, entering **10** for the dividend and **foo** for the divisor. How is your output different than what you observed in steps 2.2 and 2.3 above? Click or tap here to enter text.
   3. Change the error action preference back to its default value:  
      **$ErrorActionPreference = "Continue"**
2. Modify the code to enclose the division inside a try/catch block:

$dividend=Read-Host "Enter a dividend: "

$divisor=Read-Host "Enter a divisor: "

try{

    $quotient=$Dividend/$divisor

    Write-Host "The answer my friend is $quotient" -f green

}catch{

    Write-Host "You can't divide by 0." -f red

}

* 1. Run the script, entering **10** for the dividend and **2** for the divisor.
     1. What is the output? Click or tap here to enter text.
  2. Run the script, entering **10** for the dividend and **0** for the divisor.
     1. What was the output? Click or tap here to enter text.
     2. Did you get an error? Click or tap here to enter text.
  3. Run the script, entering **10** for the dividend and **foo** for the divisor.
     1. What was the output? Click or tap here to enter text.
     2. Does that make sense? Click or tap here to enter text.

1. Let’s see if we can handle different kinds of errors. Modify your code as follows, adding a specific exception for the divide by zero case. Then modify the default catch case.

$dividend=Read-Host "Enter a dividend: "

$divisor=Read-Host "Enter a divisor: "

try{

    $quotient=$Dividend/$divisor

    Write-host "The answer my friend is $quotient" -f green

}catch [System.DivideByZeroException]{

    Write-host "You can't divide by 0." -f red

}catch{

    Write-Host "Something went wrong." -f red

}

* 1. Run the script, entering **10** for the dividend and **2** for the divisor.
     1. What is the output? Click or tap here to enter text.
  2. Run the script, entering **10** for the dividend and **0** for the divisor.
     1. What was the output? Click or tap here to enter text.
  3. Run the script, entering **10** for the dividend and **foo** for the divisor.
     1. What was the output? Click or tap here to enter text.

1. Clean things up by including a finally block and adding some more information for the user in the other error case. The $\_ intrinsic variable in the catch block will contain the exception that caused the error. You can access its properties to understand the error in more detail. Modify your code as follows:

$dividend=Read-Host "Enter a dividend: "

$divisor=Read-Host "Enter a divisor: "

try{

    $quotient=$Dividend/$divisor

    Write-host "The answer my friend is $quotient" -f green

}catch [System.DivideByZeroException]{

    Write-host "You can't divide by 0." -f red

}catch{

    Write-Host "Something went wrong." -f red

    $\_.Exception.Message

}finally{

    Write-Host "Thanks for doing math." -f yellow

}

* 1. Run the script entering **10** for the dividend and **2** for the divisor.
     1. What is the output? Click or tap here to enter text.
  2. Run the script entering **10** for the dividend and **0** for the divisor.
     1. What was the output? Click or tap here to enter text.
  3. Run the script entering **10** for the dividend and **foo** for the divisor.
     1. What was the output? Click or tap here to enter text.

# Task 5—Making your own exceptions.

Sometimes you catch an exception, but you can’t handle it properly in your code. In that case you could create and throw your own exception.

## Steps

1. Modify your code to throw an exception whenever an error other than divide by zero occurs.

$dividend=Read-Host "Enter a dividend: "

$divisor=Read-Host "Enter a divisor: "

try{

    $quotient=$Dividend/$divisor

    Write-host "The answer my friend is $quotient" -f green

}catch [System.DivideByZeroException]{

    Write-host "You can't divide by 0." -f red

}catch{

    Throw

}finally{

    Write-Host "Thanks for doing math." -f Yellow

}

* 1. Run the script, entering **10** for the dividend and **foo** for the divisor.
     1. What was the output? Click or tap here to enter text.
  2. Did the finally block run? Click or tap here to enter text.

1. You can throw a custom message as well. Modify your code as follows:

$dividend=Read-Host "Enter a dividend"

$divisor=Read-Host "Enter a divisor"

try{

    $quotient=$Dividend/$divisor

    Write-host "The answer my friend is $quotient" -f green

}catch [System.DivideByZeroException]{

    Write-host "You can't divide by 0." -f red

}catch{

    Throw "something went wrong"

}finally{

    Write-Host "Thanks for doing math." -f Yellow

}

* 1. Run the script entering **10** for the dividend and **foo** for the divisor.
     1. What was the output? Click or tap here to enter text.

# Deliverable

Upload this document with completed answers to I-Learn Canvas.