**Error Handling in PowerShell**

There are a few different kinds of errors in PowerShell, and it can be a little bit of a minefield on occasion. There are always two sides to consider, too: how you *write* code that creates errors, and how you *handle*those errors in your own code. Let's have a look at some ways to effectively utilise the different kinds of errors you can work with in PowerShell, and how to handle them.

**Different Types of Errors**

**Terminating Errors**

**Terminating** errors are mostly an "exactly what it sounds like" package in general use. They're very similar to C#'s **Exception** system, although PowerShell uses **ErrorRecord** as its base error object. Their common characteristics are as follows:

* Trigger **try**/**catch** blocks
* Return control to the caller, along with the exception, if they're not handled using a try/catch block.

There are a couple different ways to get a terminating error in PowerShell, and each has its own small differences that make them more useful in a tricky situation.

**Throw**

First is your typical **throw** statement, which simply takes an exception message as its only input.

throw "Something went wrong"

**throw** creates an exception and an ErrorRecord (PowerShell's more detailed version of an exception) in a single, simple statement, and wraps them together. Its exceptions display the entered error message, a generic **RuntimeException**, and the errorID matches the entered exception message. The line number and position displayed from a **throw** error message point back to the **throw** statement itself. **throw** exceptions are affected by **-ErrorAction** parameters of their advanced functions.

**throw** should generally be used for simple runtime errors, where you need to halt execution, but typically only if you plan to **catch** it yourself later. There is no guarantee a user will not elect to simply ignore your error with **-ErrorAction Ignore** and continue with their lives if it is passed up from your functions.

**ThrowTerminatingError()**

Next, we have a more advanced version of the throw statement. **ThrowTerminatingError()** is a method that comes directly from PowerShell's **PSCmdlet** .NET class. It is accessible only in advanced functions via the **$PSCmdlet** variable.

using namespace System.Management.Automation;

$Exception = [Exception]::new("error message")

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

$Exception,

"errorID",

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

$TargetObject # usually the object that triggered the error, if possible

)

$PSCmdlet.ThrowTerminatingError($ErrorRecord)

As you can see, these require a little more *manual labor*. You can bypass some of this by *deliberately*using the **throw** statement and then later **catch**ing the generated **ErrorRecord** to pass it into the **ThrowTerminatingError()** method directly. This saves a decent amount of the code involved, and offloads the work to PowerShell's engine instead. However, this will still involve essentially the same work being done, without necessarily giving you a chance to specify the finer details of the error record.

**ThrowTerminatingError()** creates a terminating error (as the name implies), however *unlike* **throw** its errors are *unaffected* by **-ErrorAction** parameters applied to the parent function. It will also never reveal the code around it. When you see an error thrown via this method, it will **always** display only the line where the command was called, never the code within the command that actually threw the error.

**ThrowTerminatingError()** should generally be used for *serious* errors, where continuing to process in spite of the error may lead to corrupted data, and you **want** any specified **-ErrorAction** parameter to be ignored.

**Non-Terminating Errors**

Non-terminating errors are PowerShell's concession demanded by the fact that it is a shell that supports pipelines. Not all errors warrant halting *all* ongoing actions, and only need to inform the user that a minor error occurred, but continue processing the remainder of the items.

Unlike terminating errors, non-terminating errors:

* Do **not** trigger **try**/**catch** blocks.
* Do not affect the script's control flow.

**Write-Error**

**Write-Error** is a bit like the non-terminating version of **throw**, albeit more versatile. It too creates a complete **ErrorRecord**, which has a **NotSpecified** category and some predefined **Exception** and **ErrorCode**properties that both refer to **Write-Error**.

This is a very simple and effective way to write a non-terminating error, but it is somewhat limited in its use. While you can override its predefined default exception type, error code, and essentially everything about the generated **ErrorRecord**, **Write-Error** is similar to **throw** in that it too points directly back to the line where **Write-Error** itself was called. This can often make the error display very unclear, which is rarely a good thing.

**Write-Error** is a great way to emit straightforward, non-terminating errors, but its tendency to muddy the error display makes it difficult to recommend for any specific uses. However, as it is itself a cmdlet, it is also possible to make it create *terminating* errors as well, by using the **-ErrorAction Stop** parameter argument.

**WriteError()**

**WriteError()** is the non-terminating cousin of **ThrowTerminatingError()**, and is also available as one of the members of the automatic **$PSCmdlet** variable in advanced functions. It behaves similarly to its cousin in that it too hides the code that implements it and instead points back to the point where its parent function was called as the source point of the error. This is often preferable when dealing with expected errors such as garbled input or arguments, or simple "could not find the requested item" errors in some cases.

$Exception = [Exception]::new("error message")

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

$Exception,

"errorID",

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

$TargetObject # usually the object that triggered the error, if possible

)

$PSCmdlet.WriteError($ErrorRecord)

**Emit Errors Responsibly**

There are no true hard-and-fast rules here. If one solution works especially well for what you happen to be putting together, don't let me deter you. However, I would urge you to consider *both* the impact to your users as well as yourself or your team before selecting a single-minded approach to error creation. Not all error scenarios are created equal, so it's very rare that I would recommend using one type over all others.

**WriteError()** is generally preferred over its cmdlet form **Write-Error**, because it offers more control over how you emit the error. Both tend to be used for *expected* and technically avoidable errors that occur, but the **$PSCmdlet** method has the additional benefit of not divulging implementation details unnecessarily.

If you *expect* the error to occur sometimes, I would generally advise using **WriteError()** unless the error's presence significantly impairs your cmdlet's ability to process the rest of its input, in which case I would use **ThrowTerminatingError()**.

**throw** is generally nice as a quick way to trigger a **try**/**catch** block in your own code, but I would *also*recommend using it for: those *truly* unexpected errors, that last-ditch **catch** clause after several more specific **catch** blocks have been put in place and bypassed. At this point, a final ending **catch** block that simply triggers a **throw $\_** call to re-throw the caught exception into the parent scope and quickly pass execution back to the caller. It's a very quick and very effective way to have the error be noticed, but should usually be kept internal to the function more than anything else, except in the last scenario mentioned. Use of **Write-Error -ErrorAction Stop** may potentially also be substituted for that purpose if you personally prefer.