

La gestion des erreurs sous PowerShell.

Par Laurent Dardenne, le 03/09/2013.



Niveau		
Débutant	Avancé	Confirmé
<input type="text"/>		

A Julien D. pour sa patience.

Conçu avec Powershell v2 sous Windows Seven.

Je tiens à remercier Matthew B. et Laurent C. pour leurs relectures et remarques.

Site de l'auteur : <http://laurent-dardenne.developpez.com/>

Les fichiers sources :

<http://ottomatt.pagesperso-orange.fr/Data/Tutoriaux/Powershell/La-gestion-des-erreurs-sous-PowerShell/Sources.zip>

Chapitres

1	L'ERREUR EST HUMAINE.....	4
2	LES FLUX A DISPOSITION DANS POWERSHELL	5
2.1	ERREUR SIMPLE ET ERREUR BLOQUANTE	5
2.1.1	<i>Comportement d'une erreur</i>	<i>5</i>
2.1.2	<i>Comportement d'une erreur bloquante.....</i>	<i>6</i>
2.2	FLUX D'ERREUR.....	7
3	PRINCIPE DES EXCEPTIONS.....	7
3.1	CONCEPT D'EXCEPTION.....	9
3.1.1	<i>Usage</i>	<i>9</i>
3.2	DECLENCHER UNE ERREUR	10
3.3	DECLENCHER UNE ERREUR BLOQUANTE	10
3.3.1	<i>Propager la cause réelle d'une erreur.....</i>	<i>11</i>
4	MODIFIER LE COMPORTEMENT DE LA GESTION D'ERREUR	13
4.1	LA STRUCTURE ERRORRECORD	14
4.2	MODIFIER LE COMPORTEMENT DE WRITE-ERROR	15
4.3	MODIFIER L'AFFICHAGE DES MESSAGES D'ERREUR.....	17
4.4	MODIFIER D'AUTRES COMPORTEMENTS	19
5	LA CONSTRUCTION D'UN GESTIONNAIRE D'EXCEPTION.....	19
5.1	TRY.....	19
5.2	CATCH	20
5.2.1	<i>Accès à l'objet Exception.....</i>	<i>20</i>
5.2.2	<i>Comportement avec une erreur simple</i>	<i>20</i>
5.2.3	<i>Une construction à éviter.....</i>	<i>21</i>
5.2.4	<i>Capture par type d'exception.....</i>	<i>22</i>
5.2.1	<i>Redéclenchement d'exception.....</i>	<i>24</i>
5.2.2	<i>Gestion d'exception dans le pipeline</i>	<i>24</i>
5.2.3	<i>Comment déterminer les exceptions à capturer.....</i>	<i>25</i>
5.3	FINALLY.....	26
5.4	PROPAGATION DES EXCEPTIONS IMPREVUES	28
5.5	L'INSTRUCTION TRAP	29
6	PARAMETRES COMMUNS DE CMDLET OU DE FONCTION LIES AUX ERREURS	29
6.1	ERRORACTION	29
6.2	ERRORVARIABLE	29
6.3	WARNINGVARIABLE	30
6.4	COMMENT IMPLEMENTER CE TYPE DE VARIABLE ?	30
7	GESTIONNAIRE D'EXCEPTION GLOBALE	31
8	GENERER DES CLASSES D'EXCEPTION.....	32
8.1	TYPE D'EXCEPTION UTILISABLE	32
8.2	CREER SES PROPRES EXCEPTIONS	32
9	QUELQUES OPERATIONS RECURRENTES	34
9.1	TESTER SI UNE ERREUR A EU LIEU	34
9.2	DETERMINER LE RETOUR D'EXECUTION D'UNE COMMANDE	34
9.3	REDIRECTION D'ERREUR	35
9.4	FLUX STANDARD	37

9.4.1	<i>Les nouveaux flux sous Powershell version 3</i>	37
9.5	PROPRIETE DE FORMATAGE.....	38
10	CONCLUSION	38

1 L'erreur est humaine

La programmation étant un exercice difficile et sans remède miracle, en cas d'erreur il s'avère judicieux de déterminer ce qui ne fonctionne pas, où et pourquoi. Nous devons donc considérer une erreur comme une information importante.

On oublie le plus souvent que l'on code, pour se faciliter le travail, dans un environnement contrôlé (tous les droits, toutes les ressources disponibles). L'installation d'un script dans l'environnement de production peut révéler des erreurs imprévues, voir quelques fois insoupçonnées. Les tests étant là pour valider le respect des spécifications et les scénarios d'erreur les plus pertinents. Les autres erreurs, le plus souvent par manque de temps, ne sont pas testés et seront remontées par le mécanisme de gestion d'erreur global et corrigées au cas par cas.

Dans une gestion d'erreur on doit se préoccuper de ses propres cas d'erreurs, mais aussi de ceux des autres. Pour ces derniers, il s'agit d'informations destinées aux développeurs utilisant :

- des programmes, des cmdlets ou des scripts tiers,
- des ressources du système (plus de mémoire, plus d'espace disque,...).

Coder c'est une aventure, à vous d'éviter les chausse-trapes !

2 Les flux à disposition dans Powershell

On peut utiliser un mécanisme de traces pour retrouver le contexte ou les traitements précédant l'erreur. Dans ce cas on utilisera soit le flux *Debug* utilisé lors de la phase de mise au point, soit le flux *Verbose* qui lui permettra d'afficher des informations supplémentaires jugées optionnelles lors de l'exécution en production. Ces flux pourront être activés ou désactivés à l'aide d'une variable de préférence, cf. *About_Preference_Variable.txt*.

Je vous recommande toutefois d'utiliser un système de log plus élaboré, tel que Log4NET qui permet d'adresser plusieurs dispositifs.

Powershell propose également un flux spécifique dédié aux erreurs.

2.1 Erreur simple et erreur bloquante

Powershell utilisant le pipeline pour manipuler des collections d'objets, ses concepteurs ont décidé de scinder le comportement des erreurs. Sous Powershell une erreur est bloquante ou pas.

Les termes anglais étant respectivement *Terminating Errors* et *Non-Terminating Errors* qui peuvent se traduire par *mettre fin, ou pas, à quelque chose*.

Vos traitements détermineront si vous avez besoin de traiter leurs erreurs en tant qu'erreur simple ou en tant qu'erreur bloquante. L'usage en anglais du '*non-*' rend confus la compréhension de cette distinction, puisque sémantiquement une erreur est une erreur, je parlerais donc d'erreur et d'erreur bloquante.

2.1.1 Comportement d'une erreur

Elle n'arrête pas le traitement lors de son déclenchement, mais reste une information importante à mémoriser, elle n'est pas lié à une exception, mais plutôt à une trace d'erreur temporaire. La situation la déclenchant n'est pas considérée comme bloquante. Ceci dit par défaut on avertit tout de même l'utilisateur :

```
$DebugPreference='Continue'
Function Test-TypeErreur{
    Write-Debug "Debut"
    $process = Get-WmiObject win32_Process -computername "offline"
    Write-Debug "Suite"
    1..3|
    where-object {($_ % 2) -ne 0}|
    Foreach-Object {Write-Error "Nombre impaire interdit."}

    Write-Debug "Fin"
}
Test-TypeErreur
```

DÉBOGUER : Debut

Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT : 0x800706BA)
Au niveau de ligne : 3 Caractère : 28

DÉBOGUER : Suite

Test-TypeErreur : Nombre impaire interdit.
Au niveau de ligne : 10 Caractère : 16

Test-TypeErreur : Nombre impaire interdit.
Au niveau de ligne : 10 Caractère : 16

DÉBOGUER : Fin

Dans cet exemple, trois erreurs sont déclenchées,

- une lors de l'exécution du traitement du cmdlet Get-WmiObject,
- les deux autres par la boucle Foreach-Object.

Mais aucune n'arrête le déroulement du code de la fonction. Notez que

- le résultat de l'affichage a été tronqué afin de faciliter la lecture,
- les traces de debug sont activées.

2.1.2 Comportement d'une erreur bloquante

Dans un premier temps, elle arrête le traitement lors de son déclenchement, car elle est liée au mécanisme d'exception que nous verrons par la suite.

Dans un second temps, la situation la déclenchant est suffisamment importante pour informer le code appelant ou l'utilisateur. Ce peut être par exemple une condition interne empêchant la poursuite du traitement (une 'division par zéro') ou vous ne voulez pas que votre traitement se poursuive suite à une condition particulière, droits insuffisants, invalidation d'une règle de gestion, etc.

```
$DebugPreference='Continue'  
Function Test-TypeErreur{  
    Write-Debug "Debut"  
    $process = Get-WmiObject win32_Process -computername "offline"  
    Write-Debug "Suite"  
    1..3|  
    where-Object {($_ % 2) -ne 0}|  
    Foreach-Object {Throw "Nombre impaire interdit."}  
  
    Write-Debug "fin"  
}  
Test-TypeErreur
```

DÉBOGUER : Debut

Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT : 0x800706BA)

Au niveau de ligne : 3 Caractère : 28

DÉBOGUER : Suite

Test-TypeErreur : Nombre impaire interdit.

Au niveau de ligne : 10 Caractère : 16

Dans cet exemple modifié, une erreur est déclenchée et une erreur bloquante arrête le déroulement du code la fonction, l'affichage de la trace de debug 'Suite' ne se fait donc pas.

On constate entre les deux types d'erreur, que pour une itération dans le pipeline le déclenchement d'une erreur est un moyen simple d'informer l'utilisateur d'une erreur sur un ou des objets de la collection, tout en continuant à traiter le reste de la collection.

Par exemple l'itération sur une collection de nom de serveur afin de collecter des informations, est un parfait exemple d'utilisation d'erreur. On souhaite continuer le traitement sur le reste de la collection.

En revanche la distribution de ressources vers des serveurs d'une ferme, peut nous inciter à arrêter le traitement dès la première erreur rencontrée (l'implémentation d'un mécanisme de rollback ne sera pas traitée dans ce tutoriel).

C'est donc à vous de déterminer quand utiliser l'un ou l'autre type d'erreur.

Pour les objets dotnet, qui ne sont pas des cmdlets, leur code déclenchera toujours des erreurs bloquantes.

Pour les cmdlets ce sera soit l'un soit l'autre selon le choix de son concepteur.

2.2 Flux d'erreur

Chaque erreur est insérée dans la collection d'erreur accessible *via* la variable automatique nommée `$Error` qui est créée lors du démarrage d'une session ou d'un job.

L'affichage de cette collection renverra toutes les erreurs déclenchées depuis le démarrage de votre session Powershell. L'instruction `$Error[0]` renverra la dernière erreur ayant eu lieu.

L'approche utilisée dans le code suivant :

```
If ($Error.count -eq 0) {Write-Host "Réussite".}
```

ne doit pas être utilisée. Suite à des bugs dans le parseur de Powershell, la collection `$Error` peut contenir de fausses erreurs. La version 3 de Powershell reste concernée par ces bugs. Consulter le site MsConnect pour plus de détails.

L'usage du paramètre `ErrorVariable` présenté plus avant n'est pas concerné par cette restriction.

3 Principe des exceptions

Les API de Windows ont été créées à l'aide du langage C, à une époque où la gestion des erreurs se faisait par un code retour indiquant la nature de l'erreur tout en l'associant à un message d'erreur localisable.

Association que l'on peut retrouver avec la commande suivante :

```
net helpmsg 2182
```

```
Le service demandé a déjà été démarré.
```

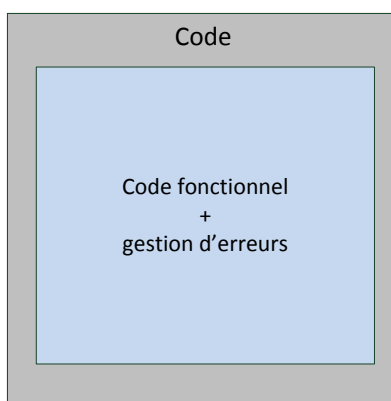
Et pour ceux de Winrm avec :

```
winrm helpmsg 0x5
```

```
Accès refusé.
```

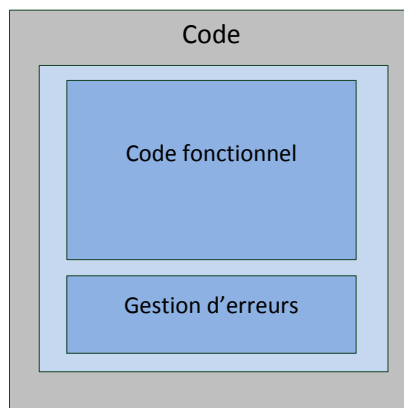
Avec cette approche, chaque appel de méthode renvoyant un code retour doit être suivi d'un test de réussite. Cette gestion est sous la responsabilité du développeur appelant ces fonctions, mais aucun mécanisme ne garantit que l'appelant prenne bien en charge le code retour.

Cette gestion d'erreur ressemble au schéma suivant, le code regroupe les traitements fonctionnels et le traitement des erreurs :



Pour faciliter l'implémentation d'une gestion d'erreur, un mécanisme d'exception a été mis en œuvre dans les langages de programmation plus récents. Les exceptions sous Powershell sont considérées comme des erreurs bloquantes et pourront être interceptées.

Dans le schéma suivant, le code de la gestion des erreurs est ici séparé du code fonctionnel :



Chaque bloc est dédié. De plus, la gestion d'erreur peut être déclarée dans le bloc de code appelant.

Voici le code du principe d'une gestion d'exception :

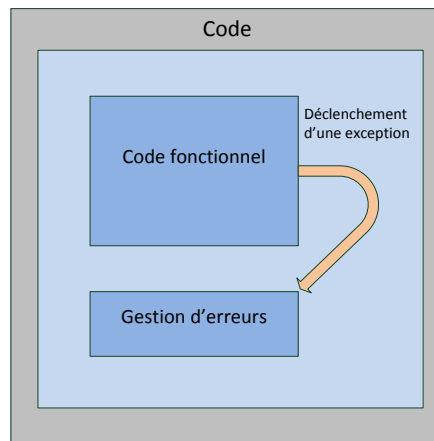
```
Try
{ ... Code Fonctionnel ... }
Catch
{ ... Gestion d'erreurs ... }
```

3.1 Concept d'exception

Comme indiqué sur le site MSDN (<http://msdn.microsoft.com/fr-fr/library/bb727317.aspx>) :

« Une des raisons pour lesquelles les exceptions ne sont pas appelées des erreurs est que le terme erreur laisse entendre en général une faute de programmation. Une exception est une violation des hypothèses implicites d'une routine ».

Par exemple, lors d'une copie, on suppose la cible existante ainsi que les droits nécessaires positionnés. Une exception est donc une condition d'erreur stoppant l'exécution du code en cours et générant une rupture de séquence :



Avantages, par rapport aux codes retour : on peut la traiter dans un bloc de code différent de celui la déclenchant et surtout, par défaut elle ne peut pas être ignorée.

Cette exception, créée par le framework dotNet, prend la forme d'un objet portant des informations sur la condition d'erreur, là où un code retour ne peut pas porter plus d'information qu'un entier, à moins de renvoyer une structure de données.

Une exception peut provenir du matériel (processeur, carte réseau), du noyau de l'OS ou le plus souvent d'une méthode objet ou d'une API.

Note : Pour les schémas, j'ai repris la présentation de l'ouvrage "Oracle PL/SQL Programming" de Steven Feuerstein.

3.1.1 Usage

La mécanique d'exception permet d'écrire du code sans se soucier des erreurs pouvant survenir (un fichier inexistant, pas de droits sur la ressource ciblée, plus de mémoire). On code les traitements sans vérifier si chaque instruction, pouvant déclencher une exception, a réussi. Ce

type d'écriture facilite la relecture du code, car il sépare le code fonctionnel du code de gestion des erreurs.

A la différence des API win32, le framework dotnet, et donc Powershell, utilise un mécanisme d'exception. Le framework permet de déclencher des exceptions personnalisées à l'aide d'une classe **Exception**, et de ses descendants, mais aussi d'intervenir si besoin sur l'exécution du mécanisme d'exception à l'aide de mots-clé du langage.

A l'aide de ceux-ci on peut par exemple capturer l'exception afin de corriger la condition d'erreur ou sauvegarder les données avant l'arrêt inopiné du programme. Une fois ceci fait on peut redéclencher l'exception pour terminer le programme.

Qui dit gestion d'exception dit gestionnaire d'exception. Son rôle est d'intercepter une, plusieurs ou toutes les exceptions déclenchées dans un script, une fonction ou un bloc de code.

3.2 Déclencher une erreur

Pour générer une erreur, on utilisera le cmdlet **Write-Error**. Celui-ci attend, dans sa forme la plus simple, une chaîne de caractères :

Exemple :

```
$Tableau=@('Un', 'Deux')
$Name='Trois'
if ($Tableau -notContains $Name)
{ Write-Error "L'élément '$Name' n'existe pas."}
if ($Tableau -notContains $Name)
{ Write-Error "L'élément '$Name' n'existe pas." : L'élément 'Trois' n'existe pas.
```

Dans une console, le résultat d'un appel à **Write-Error** s'affiche par défaut à l'écran.

3.3 Déclencher une erreur bloquante

Pour générer une erreur bloquante, on déclenchera une exception à l'aide de l'instruction **Throw**. Celle-ci attend dans sa forme la plus simple une chaîne de caractères :

```
$Name=' '
if ($Name -eq [string]::Empty)
{ Throw "Le paramètre 'Name' doit être renseigné."}
throw : Le paramètre 'Name' doit être renseigné.
```

Ici la classe de l'exception déclenchée sera *System.Management.Automation.RuntimeException*.

On peut également lui passer en paramètre une instance d'objet de la classe *Exception* ou dérivée de cette classe :

```
if (-not (Test-Path Function:\TestInconnue))
{
    $ObjetException=New-Object System.NotImplementedException(
        "La fonction TestInconnue n'existe pas.")
    Throw $ObjetException
}
```

throw : La fonction `TestInconnue` n'existe pas.

Ici la classe de l'exception déclenchée sera `System.NotImplementedException`.

Note : La variable automatique `$MaximumErrorCount` permet de modifier la taille maximum de la collection `$Error`. Une fois la taille maximum atteinte, Powershell supprimera l'erreur la plus ancienne afin d'insérer la plus récente en début de collection.

3.3.1 Propager la cause réelle d'une erreur

Le plus souvent le message associé à une erreur est une explication de l'erreur :

```
Try {
    $msg="L'argument n'est pas renseigné."
    $Exception=New-Object System.ArgumentException($Msg,'Name')
    throw $Exception
}
Catch {
    throw "Traitement en erreur : $($_.Exception.Message) "
```

throw : Traitement en erreur : L'argument n'est pas renseigné.
Nom du paramètre : Name

Cette approche ne permet pas de retrouver le contexte de l'erreur, c'est-à-dire la cause de son déclenchement porté par la première exception :

```
$Error[0].Exception.InnerException -eq $null
True
```

En revanche celle-ci le permet :

```
Try {
    $msg="L'argument n'est pas renseigné."
    $Exception=New-Object System.ArgumentException($Msg,'Name')
    throw $Exception
}
Catch [System.ArgumentException] {
    $Msg="Traitement en erreur :"+ $_.Exception.Message
    $Exception=New-Object System.ApplicationException($Msg,$_.Exception)
    throw $Exception
}
```

Ainsi l'exception d'origine est propagée, ce qui évite de parcourir la collection `$Error` afin de reconstruire l'enchaînement des erreurs. A la différence de la construction précédente, ici la propriété `InnerException` est renseignée :

```
$Error[0].Exception.InnerException -eq $null
False
$Error[0].Exception.InnerException
L'argument n'est pas renseigné.
Nom du paramètre : Name
```


4 Modifier le comportement de la gestion d'erreur

La variable de préférence **\$ErrorActionPreference** paramètre le comportement de la gestion des erreurs. Ses valeurs valides sont :

<i>Stop</i> (Arrêter)	Affiche le message d'erreur et arrête l'exécution.
<i>Inquire</i> (Demander)	Affiche le message d'erreur et vous demande si vous souhaitez continuer. <u>Attention</u> ce paramétrage ne pourra être utilisé lors de l'exécution d'un script dans un Job ou dans une console Orchestrator.
<i>Continue</i> (Continuer) Valeur par défaut	Affiche le message d'erreur et continue l'exécution lors d'un appel à <i>Write-Error</i> ou stop l'exécution lors d'un appel à <i>Throw</i> .
<i>SilentlyContinue</i> (Continuer en mode silencieux)	Aucune erreur n'arrête l'exécution. Le message n'est pas affiché sur la console, l'exécution continue sans interruption. <u>La variable <i>\$Error</i> est renseignée.</u>

La portée de cette variable automatique est locale.

Reprenons l'exemple déclenchant des erreurs et modifions le comportement :

```
$ErrorActionPreference='Stop'
Function Test-TypeErreur{
    #Pour le test, décommentez la ligne suivante
    #$ErrorActionPreference="SilentlyContinue" # portée LOCALE
    write-Host "Debut"
    $process = Get-WmiObject win32_Process -computername "offline"
    write-Host "Suite"
    1..3|
        where-Object {($_ % 2) -ne 0}|
        Foreach-Object {Write-Error "Nombre impaire interdit."}
    write-Host "Fin"
}
Test-TypeErreur

Debut
Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT : 0x800706BA)
Au niveau de ligne : 3 Caractère : 28
```

Avec ce paramétrage, le code de la fonction est stoppé dès la création d'une erreur sans distinction de catégorie.

On peut aussi ne pas tenir compte des erreurs bloquantes (des exceptions), en utilisant la valeur *'SilentlyContinue'* :

```
$Error.Clear()
$ErrorActionPreference="SilentlyContinue"
Function Test-TypeErreur{
    Write-Host "Debut"
    $process = Get-WmiObject Win32_Process -computername "offline"
    Write-Host "Suite"
    1..3|
        where-Object {($_ % 2) -ne 0}|
        Foreach-Object {Throw "Nombre impair interdit."}

    Write-Host "Fin"
}
Test-TypeErreur
```

L'exécution de cette fonction affichera :

```
Debut
Suite
Fin
```

Je trouve cet usage plutôt inhabituel, notez que la collection d'erreurs contient bien toutes les erreurs créées, bien que celles-ci ne soient pas affichées :

```
$Error
throw : Nombre impair interdit.
Au niveau de ligne : 7 Caractère : 26
+   Foreach-Object {Throw <<<< "Nombre impair interdit."}...

throw : Nombre impair interdit.
Au niveau de ligne : 7 Caractère : 26
+   Foreach-Object {Throw <<<< "Nombre impair interdit."}...

Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT : 0x800706BA)
Au niveau de ligne : 3 Caractère : 28
+   $process = Get-WmiObject <<<< Win32_Process -computername "offline" ...
```

Voir aussi les exemples de l'aide en ligne :

```
Get-Help Preference_Variable
```

4.1 La structure *ErrorRecord*

L'analyse d'une erreur nécessite de connaître la classe *ErrorRecord*, vous retrouverez le détail de ses membres sur le lien suivant :

http://msdn.microsoft.com/fr-FR/library/system.management.automation.errorrecord_members.aspx

4.2 Modifier le comportement de Write-Error

Si la variable `$ErrorActionPreference` est assignée à `Stop`, un appel à `Write-Error` déclenchera une exception du type `System.Management.Automation.ActionPreferenceStopException` :

```
$Error.Clear()
$ErrorActionPreference="Stop"
Function Test-Error {Write-Error "Test"}
Test-Error
Test-Error : Test
```

L'affichage de la dernière erreur renvoie le texte du message d'erreur :

```
$Error[0]
L'exécution de la commande s'est arrêtée, car la variable de préférence « ErrorActionPreference » ou le paramètre courant a la valeur Stop : Test
```

Affichons le type de l'erreur :

```
$Error[0].GetType().FullName
System.Management.Automation.ActionPreferenceStopException
```

L'affichage des propriétés de la dernière erreur renvoie les détails suivants :

```
$Error[0] | select *
ErrorRecord      : Test
WasThrownFromThrowStatement : False
Message          : L'exécution de la commande s'est arrêtée, car la variable de préférence « ErrorActionPreference » ou le paramètre courant a la valeur Stop : Test
Data             : {}
InnerException    :
TargetSite       : System.Collections.ObjectModel.Collection`1[System.Management.Automation.PSObject] Invoke(System.Collections.IEnumerable)
HelpLink         :
Source           : System.Management.Automation
```

On constate que l'erreur contient un champ `ErrorRecord` détaillant l'erreur générée par l'appel à `Write-Error`. Son affichage renverra :

```
$Error[0].ErrorRecord | select *
PSMessageDetails :
Exception        : Microsoft.PowerShell.Commands.WriteErrorException: Test
TargetObject     :
CategoryInfo     : NotSpecified: (:) [Write-Error], WriteErrorException
FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException,Test-Error
ErrorDetails     :
InvocationInfo   : System.Management.Automation.InvocationInfo
PipelineIterationInfo : {0, 0}
```

A son tour, l'affichage de la propriété imbriquée `Exception` renvoie les détails suivants :

```
$Error[0].ErrorRecord.Exception | select *
Message      : Test
Data         : {}
InnerException :
TargetSite   :
StackTrace   :
HelpLink     :
Source       :
```

Pour cet exemple, le détail porte peu d'informations.

Pour obtenir des informations sur le contexte de l'erreur, on affiche le contenu de la propriété *InvocationInfo* :

```
$Error[0].ErrorRecord.InvocationInfo |select *
MyCommand      : Test-Error
BoundParameters : {}
UnboundArguments : {}
ScriptLineNumber : 1
OffsetInLine    : 11
HistoryId       : 17
ScriptName      :
Line            : Test-Error
PositionMessage :
    Au niveau de ligne : 1 Caractère : 11
    + Test-Error <<<<
InvocationName  : Test-Error
PipelineLength  : 1
PipelinePosition : 1
ExpectingInput  : False
CommandOrigin   : Runspace
```

Selon le contexte d'exécution, on peut donc connaître le nom du script, le nom de la fonction et le numéro de la ligne provoquant l'erreur.

Sous Powershell version 3, la classe *ErrorRecord* propose un nouveau membre :

```
$Error[0].StackTrace
```

En revanche si on modifie la variable *\$ErrorActionPreference* afin d'intercepter l'exception dans un bloc catch, le contenu diffère :

```
$Error.Clear()
$ErrorActionPreference="Stop"
Function Test-Error {
    Try {
        Write-Error "Test"
    } catch [System.Management.Automation.ActionPreferenceStopException]{
        Write-Warning "Erreur gérée"

        Write-Host "`r`n----- Détail exception courante`r`n"
        $_
        $_.GetType().FullName
        $_.ErrorRecord|select * #Aucun affichage
        $_|select * #Aucun affichage
    }
}
Test-Error
```

Dans ce cas, la variable *\$_* est toujours une instance de la classe *ErrorRecord* hébergeant une instance d'exception.

Affichons le type de l'erreur :

```
$Error[0].GetType().FullName  
System.Management.Automation.ErrorRecord
```

L'erreur contenue dans `$Error[0]` est bien l'erreur générée par l'appel à **Write-Error**, mais elle ne contient plus ici les informations de l'exception `ActionPreferenceStopException` :

```
$Error[0].ErrorRecord -eq $null  
True
```

Ce comportement est identique pour toutes les erreurs simples stoppées. Selon le cas on accèdera aux informations de l'erreur soit par :

```
$Error[0].ErrorRecord.Exception
```

Exemple :

```
$Error.Clear()  
$ErrorActionPreference="Stop"  
Dir x:\ #Suppose le lecteur X inexistant  
$Error[0]  
$Error[0].ErrorRecord|select *  
$Error[0].ErrorRecord.Exception|select *
```

Soit par :

```
$Error[0].Exception
```

Exemple :

```
$Error.Clear()  
$ErrorActionPreference="Continue"  
Dir x\  
$Error[0]  
$Error[0]|select *  
$Error[0].Exception|select *
```

4.3 Modifier l'affichage des messages d'erreur

La variable de préférence `$ErrorView` paramètre le format d'affichage des messages d'erreur dans Windows PowerShell. Ses valeurs valides sont :

<i>NormalView</i>	Une vue détaillée conçue pour la plupart des utilisateurs.
Valeur par défaut	Comprend une description de l'erreur, le nom de l'objet impliqué dans l'erreur et des flèches (<<<<<) qui pointent vers les mots de la commande qui ont provoqués l'erreur.
<i>CategoryView</i>	Une vue succincte, structurée, conçue pour les environnements de production. Le format est : {Category}: ({TargetName}:{TargetType}):[{Activity}],{Reason}

Voir aussi les exemples de l'aide en ligne :

```
Get-Help Preference_variable
```

Cette possibilité nécessite une construction plus élaborée du message d'erreur et implique une réflexion sur la catégorisation des erreurs de chacun de vos traitements :

```
...} catch {
    #PSCmdlet est une variable automatique créée
    #dans la portée d'une fonction avancée.
    $PSCmdlet.WriteError(
        (New-Object System.Management.Automation.ErrorRecord(
            $_.Exception,
            "PDFDocumentInfoUriOrPath",
            "InvalidOperation",
            ("{0} : {1}" -f $MyInvocation.Mycommand.Name, $_.Exception.Message))
        )
    )
}
finally {...
```

Le cmdlet Write-Error autorise également ce type de construction :

```
$Message="{0} : {1}" -f $MyInvocation.Mycommand.Name, $_.Exception.Message
Write-Error -Message $Message `
    -Exception $_.Exception `
    -Category InvalidOperation ` #Catégorisation Normée, voir le SDK
    -TargetObject $MyObject `
    -RecommendedAction "Corriger le chemin d'accès." `
    -ErrorId "CantWriteLog, Measure-Time" `
    -CategoryReason PDFDocumentInfoUriOrPath #Catégorisation personnelle
```

Pour plus d'informations consultez "Class ErrorCategoryInfo " dans le Kit de développement logiciel (SDK) Windows PowerShell. Vous trouverez un exemple complet d'un tel usage dans le script *Replace-String.ps1*.

Comme indiqué dans la page suivante :

<http://msdn.microsoft.com/en-us/library/windows/desktop/ms714465%28v=vs.85%29.aspx>

pour les erreurs simples, le message affiché est celui du champ exception. Pour le remplacer utilisez la propriété *ErrorDetails* :

```
$ErrRec=New-Object System.Management.Automation.ErrorRecord(
    $_.Exception,
    "PDFDocumentInfoUriOrPath",
    "InvalidOperation",
    ("{0} : {1}" -f $MyInvocation.Mycommand.Name, $_.Exception.Message)
)
$ErrRec.ErrorDetails=New-Object System.Management.Automation.ErrorDetails(
    "Mon message de remplacement")
$PSCmdlet.WriteError($ErrRec)
```

4.4 Modifier d'autres comportements

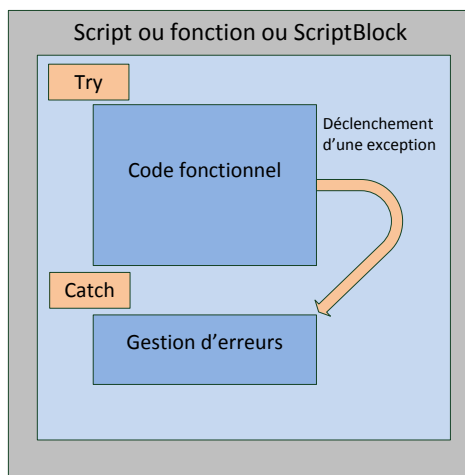
Pour la version 2 de Powershell, la modification de comportement peut se faire sur les flux suivants :

```
dir variable:*preference
```

Name	Value
-----	-----
ConfirmPreference	High
DebugPreference	SilentlyContinue
ErrorActionPreference	SilentlyContinue
ProgressPreference	Continue
VerbosePreference	SilentlyContinue
WarningPreference	Continue
WhatIfPreference	False

5 La construction d'un gestionnaire d'exception

La construction d'un gestionnaire d'exception nécessite l'usage des instructions *Try*, *Catch* et *Finally* :



5.1 Try

Cette instruction débute la gestion d'exception pour un bloc de code que vous jugez utile de protéger. Chaque déclenchement d'exception forcera l'exécution du code dans le(s) bloc(s) *Catch* ou *Finally* associé(s). Un bloc *Try* seul n'a donc pas de raison d'être et provoquera une erreur de parsing. Le code en dehors de cette construction ne sera pas protégé des exceptions.

Exemple :

```
Try
{ code d'un traitement à protéger }
Catch
{ code de gestion d'erreur }
```

Le code à protéger peut être constitué d'une seule ligne pouvant être un appel à un cmdlet ou encore un appel à une méthode d'un objet.

Il reste possible de placer du code en dehors du bloc try :

```
... code sans protection
Try
{ code d'un traitement à protéger }
Catch
{ code de gestion d'erreur }
... suite du code sans protection
```

Vous seul décidez des sections de code que vous voulez protéger par ce mécanisme.

5.2 Catch

Cette instruction débute un bloc de code de traitement d'un ou plusieurs types d'exceptions, déclenchées dans le bloc *Try* associé. Un bloc *Catch* seul n'a pas de raison d'être et provoquera une erreur de parsing.

Un bloc *Try* peut être associé à plusieurs blocs *Catch*. Chaque bloc permettra de traiter (capturer) un ou plusieurs types d'exception.

5.2.1 Accès à l'objet Exception

Dans le bloc de code associé à l'instruction *Catch*, l'objet exception est accessible *via* la variable automatique `$_`.

5.2.2 Comportement avec une erreur simple

La construction Try/Catch n'opère que sur les erreurs bloquantes, c'est à dire que sur un déclenchement d'exception. Dans l'exemple suivant l'erreur n'est pas prise en charge par le bloc catch :

```
$ErrorActionPreference="Continue"
Try {
    $process = Get-WmiObject win32_Process -computername "offline"
    write-host "Instruction suivante"
} Catch [System.Management.Automation.ActionPreferenceStopException]{
    write-warning "Erreur gérée."
}
write-host "Suite du script"

Get-WmiObject : Le serveur RPC n'est pas disponible. (Exception de HRESULT : 0x800706BA)
Instruction suivante
Suite du script
```

Si la variable `$ErrorActionPreference` est assignée à *Stop*, dans ce cas une erreur déclenchera une exception du type `System.Management.Automation.ActionPreferenceStopException` :

```

$ErrorActionPreference="Stop"
Try {
    $process = Get-WmiObject win32_Process -computername "offline"
    Write-Host "Instruction suivante"
} Catch [System.Management.Automation.ActionPreferenceStopException]{
    Write-Warning "Erreur gérée."
}
Write-Host "Suite du script"

```

AVERTISSEMENT : Erreur gérée.

Suite du script

On peut également utiliser le paramètre **-ErrorAction** si on souhaite traiter les erreurs au cas par cas, c'est à dire sur un cmdlet et pas sur la totalité du bloc de code protégé :

```

$ErrorActionPreference="Continue"
Try {
    $process = gwmi win32_Process -computer "offline" -ErrorAction Stop
    Write-Host "Instruction suivante"
} Catch [System.Management.Automation.ActionPreferenceStopException]{
    Write-Warning "Erreur gérée."
}
Write-Host "Suite du script"

```

AVERTISSEMENT : Erreur gérée.

Suite du script

5.2.3 Une construction à éviter

Le code suivant est à éviter :

```

$ErrorActionPreference="Continue"
Try {
    $process = gwmi win32_Process -computer "offline" -ErrorAction Stop
    Write-Host "Instruction suivante"
} Catch {
    #traitement d'une erreur Stop
    Write-Warning "Traitement de l'erreur."
}

```

Ici ce gestionnaire d'exception traitera toutes les exceptions déclenchées dans le bloc *try*. Le traitement d'une exception due à une insuffisance de droit ou à un manque de mémoire, n'est pas concerné par ce gestionnaire.

Vous devez tant que faire se peut traiter précisément les exceptions déclenchées par vos traitements. Cette approche considère que toutes les exceptions se valent, ce qui est faux.

Cette construction est donc le plus souvent un abus du langage.

Je vous laisse réfléchir sur l'utilité et la pertinence de la construction suivante :

```
$ErrorActionPreference="Continue"
Try {
    $process = gwmi win32_Process -computer "offline" -ErrorAction Stop
    Write-host "Instruction suivante"
} Catch {}
#Suite du code
```

En consultant le site <http://fr.thedailywtf.com/> vous y lirez que tout est possible ;-)

5.2.4 Capture par type d'exception

On peut intercepter une exception par son nom de classe, dans ce cas on ordonnera les noms de classe d'exception du plus particulier au plus général. Exemple :

```
$sb={
    try {
        $i=0 ; $chemin = "chemin_inexistant"
        #5/$i #Déclenche l'exécution du bloc catch "[exception] non spécifié"
        [io.directory]::SetCurrentDirectory($chemin)
    }
    catch [System.IO.DirectoryNotFoundException] {
#catch [System.IO.IOException]
        write-warning "[exception] $($_.Exception.GetType().FullName)"
    }
    catch { write-warning "[exception] non spécifié" }
    finally { "Fin du traitement de try/catch" }
}
&$sb
```

Si on remplace, le nom de classe *System.IO.DirectoryNotFoundException* d'exception par une classe ancêtre :

```
catch [System.IO.IOException] {
```

Le code fonctionnera toujours, mais dans ce cas le bloc **Catch** couvrira les exceptions suivantes :

Classe	Description
System.IO.DirectoryNotFoundException	Exception déclenchée lorsqu'une partie d'un fichier ou d'un répertoire est introuvable.
System.IO.DriveNotFoundException	Exception déclenchée lors d'une tentative d'accès à un lecteur ou partage qui n'est pas disponible.
System.IO.EndOfStreamException	Exception déclenchée en cas de tentative de lecture au-delà de la fin du flux.
System.IO.FileLoadException	Exception déclenchée lorsqu'un assembly managé est trouvé mais ne peut pas être chargé.
System.IO.FileNotFoundException	Exception déclenchée lors d'une tentative d'accès à un fichier qui n'existe pas sur le disque, échoue.
System.IO.PathTooLongException	Exception déclenchée lorsqu'un nom de chemin d'accès ou de fichier est plus long que la longueur maximale définie par le système.

Qui représentent, par défaut, toutes les classes dérivées de *System.IO.IOException*.

Ce qui implique que dans la construction suivante :

```
catch [System.IO.IOException] { "IOException" }
catch [System.IO.DirectoryNotFoundException] {"Inutile "}
```

Le second bloc ne sera jamais déclenché, car le second type d'exception, *DirectoryNotFoundException*, sera traité par le premier bloc **Catch**.

Le parseur renvoie donc une exception sur ce type de construction :

System.IO.DirectoryNotFoundException : Le type d'exception *System.IO.DirectoryNotFoundException* est déjà géré par un précédent gestionnaire.

En revanche l'inverse est possible :

```
catch [System.IO.DirectoryNotFoundException] {"Directory Not Found "}
catch [System.IO.IOException] { "Autre IOException" }
```

Car l'ordre des exceptions, la plus particulière en premier suivi des plus générales, est respecté.

Enfin, il est possible d'associer plusieurs types d'exception à un seul bloc catch :

```
catch [IO.DirectoryNotFoundException], [IO.FileNotFoundException]
{ "Problème de chemin d'accès." }
```

5.2.1 Redéclenchement d'exception

Pour redéclencher une exception, on procédera ainsi :

```
try {
    write-Host "Initialisation des ressources."
    Throw (New-Object System.ApplicationException("Erreur applicative.") )
}
catch {
    write-warning "Libération des ressources."
    write-Host "`t Erreur ! Redéclenche l'exception."
    Throw $_
}
```

Affichera :

```
Initialisation des ressources.
AVERTISSEMENT : Libération des ressources.
    Erreur ! Redéclenche l'exception.
throw : Erreur applicative.
Au niveau de ligne : 3 Caractère : 8
+ Throw <<<< (New-Object System.ApplicationException("Erreur applicative.") )
+ CategoryInfo          : OperationStopped: (:) [], ApplicationException
+ FullyQualifiedErrorId : Erreur applicative.
```

Notez que le contexte de cette exception redéclenchée restera lié au contexte de l'exception d'origine, c'est-à-dire le cas d'exception « Erreur applicative », et pas à celui du bloc où elle est redéclenchée. Plus précisément le contenu de la propriété `Invocation` restera lié au contexte de l'exception déclenchée dans le bloc `try` et pas à celui de l'exception redéclenchée dans le bloc `catch`.

On utilise ici la construction devant être évitée, la différence est qu'on libère les ressources puis on renvoie l'exception vers les gestionnaires d'exception de l'appelant qui pourraient la traiter.

Note :

On peut parfois vouloir redéclencher une exception en modifiant son message tout en gardant le type de l'exception. Dans ce cas on doit tenir compte des classes d'exception d'accès privé. Pour résoudre ce problème on utilisera la fonction déclarée dans le script `New-Exception.ps1`, disponible dans les sources.

5.2.2 Gestion d'exception dans le pipeline

La gestion des exceptions dans un enchaînement d'instructions utilisant le pipeline n'est pas aisée à réaliser, consultez ce post, [ainsi que les commentaires](#) pour plus de détail :

<http://powershell.com/cs/blogs/tobias/archive/2010/01/01/cancelling-a-pipeline.aspx>

La version 3 de powershell propose l'exception interne `StopUpstreamCommandsException`, voir cette demande de modification de l'accès de cette API :

<https://connect.microsoft.com/PowerShell/feedback/details/768650/enable-users-to-stop-pipeline-making-stopupstreamcommandsexception-public>

5.2.3 Comment déterminer les exceptions à capturer

La difficulté est de connaître toutes les exceptions qu'une méthode peut déclencher, certaines sont documentées sur le site MSDN. En revanche la documentation des cmdlets Powershell ne référence pas les exceptions pouvant être déclenchées par un cmdlet. Vous devrez donc, à l'aide de jeux de tests déterminer celles que vous jugerez utile de gérer dans vos traitements.

Ce jeu de test va valider votre gestion d'exception.

Dans un exemple précédent, on utilise la méthode suivante :

```
[IO.Directory]::SetCurrentDirectory($chemin)
```

Pour retrouver les exceptions que cette méthode peut déclencher on consulte le site MSDN :

<http://msdn.microsoft.com/fr-fr/library/system.io.directory.setcurrentdirectory%28v=vs.80%29.aspx>

Dans la seconde partie du document intitulé « *LaNotionDobjetSousPowerShell.pdf* », disponible dans les sources, vous trouverez quelques informations sur l'usage du site MSDN. Je n'y ai pas modifié les recopies d'écran, car ce site est régulièrement modifié, mais les principes de navigation restent identiques. En revanche selon votre version de Powershell, pensez à modifier le framework ciblé. Pour la version 2 ciblez le framework 2.0, pour la version 3 le framework 4.0 et pour la version 4.0 le framework 4.5.

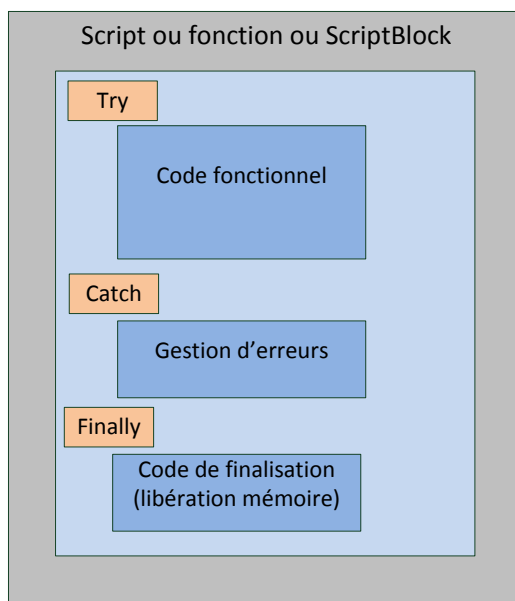
Dans le cas où vous ne disposez pas de la documentation des API que vous utilisez, vous pouvez retrouver les noms des exceptions à l'aide de la fonction *Resolve-Error*.

Voir aussi cet article :

<http://mohundro.com/blog/2009/01/26/finding-the-right-exception-to-throw/>

5.3 Finally

Cette instruction débute un bloc de code qui sera exécuté dans tous les cas, qu'il y ait eu ou non une erreur. Son rôle est d'assurer de l'exécution d'un code de libération de ressources ou d'un traitement final :



C'est dans ce bloc qu'on appellera la méthode *Dispose()* sur des objets ayant des ressources système à libérer ou pour clore des connexions à des ressources externes:

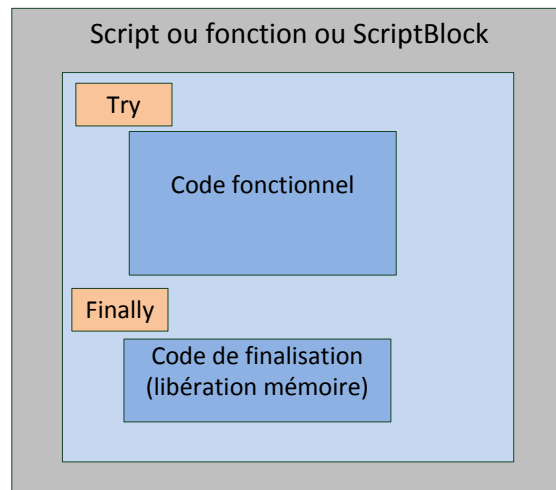
```
try {
    $Stream = New-Object System.IO.MemoryStream
    $isDisposable=$Stream -is [System.IDisposable]
    if ($isDisposable)
    { write-warning "L'objet doit être libéré explicitement." }
}
finally
{
    if ($Stream -ne $Null)
    {
        write-Debug 'Libération des ressources système de l'objet $Stream.'
        $Stream.Dispose()
        $Stream=$null
    }
}
```

Notez le test sur le contenu de la variable qui évite de déclencher une erreur si celui-ci n'est pas renseigné.

On ne peut associer qu'une seule instruction *Finally* à une instruction *Try*.

Un bloc *Finally* seul n'a pas de raison d'être et provoquera une erreur de parsing.

La construction suivante est autorisée :



Elle permet d'implémenter un code de finalisation tout en laissant au code appelant la responsabilité de traiter les exceptions que votre traitement pourrait déclencher.

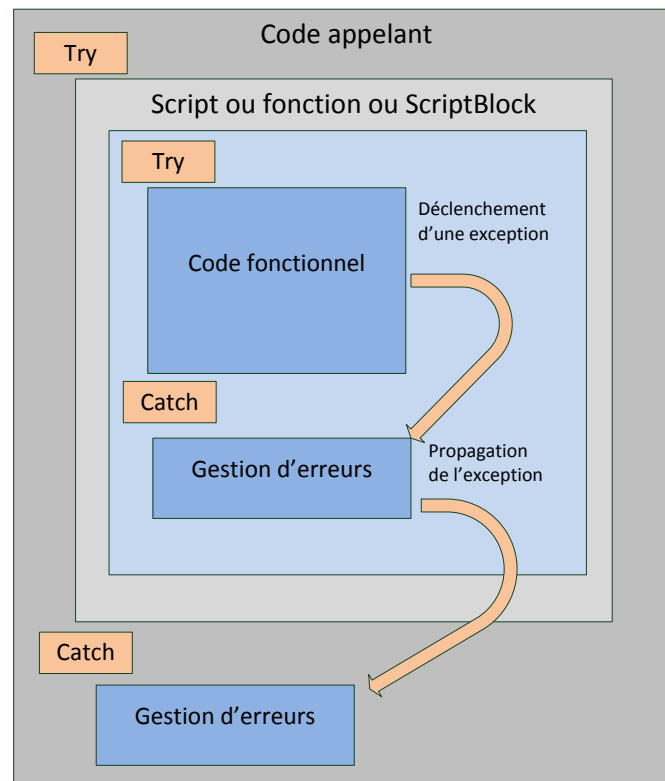
On peut donc décider de ne pas traiter localement l'exception, mais de la propager.

Voir aussi le fichier d'aide :

`Get-help about_Try_Catch_Finally`

5.4 Propagation des exceptions imprévues

Nous avons vu précédemment qu'une exception déclenchée dans un bloc de code protégé par l'instruction *Try* était gérée dans le bloc *Catch* associé, mais si celui-ci ne gère pas le type de cette exception (sous réserve de respecter le principe de codage énoncé dans la chapitre 'Une construction à éviter) le mécanisme d'exception la propage au bloc appelant et ainsi de suite jusqu'à trouver un gestionnaire d'exception pouvant la traiter :



S'il n'existe aucun gestionnaire d'exception, le host étant le niveau le plus haut, c'est dans celui-ci que l'exception sera affichée afin de nous informer d'une erreur fatale. Sous réserve que le host dispose d'une zone d'affichage, ce qui n'est par exemple pas le cas du host d'Orchestrator 2012.

Ainsi nous serons averti de l'existence d'une erreur à corriger, qui peut être une ressource nécessaire à mettre à disposition (prérequis), par exemple démarrer un service ou une VM hébergeant un serveur, utiliser un compte avec suffisamment de droits, etc.

Note :

En cas de déclenchement d'une exception lors de l'exécution d'un script Powershell exécuté dans un processus externe, Powershell renverra un code retour (%ErrorLevel%) égal à 1.

5.5 L'instruction *Trap*

L'instruction *Trap* est équivalente aux instructions Try-Catch-Finally, ayant une préférence pour ces dernières je ne la traiterais pas ici.

Sur le sujet vous pouvez consulter ces liens :

<http://blogs.msdn.com/b/powershell/archive/2009/06/17/traps-vs-try-catch.aspx>

<http://blogs.msdn.com/b/powershell/archive/2006/12/29/documenting-trap-and-throw.aspx>

<http://blogs.msdn.com/b/powershell/archive/2006/04/25/583234.aspx>

Selon les cas, elle peut faciliter un mécanisme de reprise sur erreur.

6 Paramètres communs de cmdlet ou de fonction liés aux erreurs

Chaque cmdlet peut modifier le fonctionnement par défaut des erreurs à l'aide du paramètre commun *ErrorAction*. Quant au paramètre commun *ErrorVariable* il permet de récupérer les erreurs déclenchées par le cmdlet ou la fonction et seulement les siennes.

6.1 *ErrorAction*

Ce paramètre détermine comment l'applet de commande répond à une erreur en remplaçant la valeur de la variable *\$ErrorActionPreference* pour la commande actuelle. Ses valeurs valides sont les mêmes que celles de la variable de préférence *\$ErrorActionPreference*.

Notez que la version 3 de Powershell autorise pour ce paramètre une valeur supplémentaire nommée *Ignore*. Cette valeur indique que les erreurs ne seront pas affichées ni ajoutées à la collection *\$Error*

En rappel du principe que l'erreur est humaine, voir le bug suivant :

<http://connect.microsoft.com/PowerShell/feedback/details/763621/erroraction-ignore-is-broken-for-advanced-functions>

6.2 *ErrorVariable*

Ce paramètre mémorise, dans le nom de variable spécifiée ainsi que dans la variable automatique *\$Error*, les messages d'erreur déclenchés par un cmdlet.

Par défaut, les nouveaux messages d'erreur remplacent ceux déjà stockés dans la variable. Pour ajouter le message d'erreur au contenu de la variable, saisissez un signe plus (+) avant le nom de variable.

Par exemple, la commande suivante crée la variable *\$MyErrors*, puis y stocke toutes les erreurs :

```
get-process -id 6 -ErrorVariable MyErrors
```

La commande suivante complète la variable *\$MyErrors* avec les nouvelles erreurs :

```
get-process -id 2 -ErrorVariable +MyErrors
```

6.3 WarningVariable

Ce paramètre mémorise, dans le nom de variable spécifiée, les messages d'avertissement générés par un cmdlet. Une fois ce paramètre précisé les avertissements ne sont plus affichés :

```
function Test-warning {  
    [CmdletBinding()]  
    param()  
        write-warning "Avertissement !"  
}  
  
$WarningActionPreference="Continue"  
Remove-Variable MesWarnings -ErrorAction SilentlyContinue  
Test-warning -warningvariable MesWarnings  
$MesWarnings
```

Avertissement !

Un autre exemple lié à Office 365 :

```
Import-Module MSOnline  
Connect-MsolService -credential $Credentials -warningvariable MsolWarn  
$MsolWarn
```

Ici la variable *MsolWarn* peut contenir un avertissement concernant la disponibilité d'une nouvelle version du module MSOnline.

Ou encore sous Exchange :

```
$MBStat=Get-MailboxStatistics -Identity $Id -warningvariable gmbxswarning  
-warningaction stop
```

Si la boîte mail ciblée ne contient pas d'élément, le cmdlet *Get-MailboxStatistics* ne renvoie pas d'information, mais un warning. Selon le traitement on peut choisir de générer une erreur bloquante à l'aide du paramètre *WarningAction*. L'exception déclenchée sera du type

System.Management.Automation.ActionPreferenceStopException

6.4 Comment implémenter ce type de variable ?

Vous trouverez une implémentation sur le lien suivant :

http://powershell-scripting.com/index.php?option=com_joomlaboard&Itemid=76&func=view&id=15033&catid=14

7 Gestionnaire d'exception globale

Dans le chapitre «Propagation des exceptions imprévues» nous avons vu qu'une exception peut ne pas être gérée dans le code, ce qui stoppera le script en cours. La mise en place d'un mécanisme de mémorisation de ce type d'erreur sera très utile aux personnes du support.

La console Powershell est exécutée dans un domaine d'application dotnet, qui propose un événement dédié à cette gestion :

```
[System.AppDomain]::CurrentDomain|gm -MemberType Event
TypeName: System.AppDomain
Name      MemberType Definition
-----
AssemblyLoad Event      System.AssemblyLoadEventHandler
...
UnhandledException Event      System.UnhandledExceptionEventHandler ...
```

Je préfère toutefois créer une fonction implémentant un tel gestionnaire.

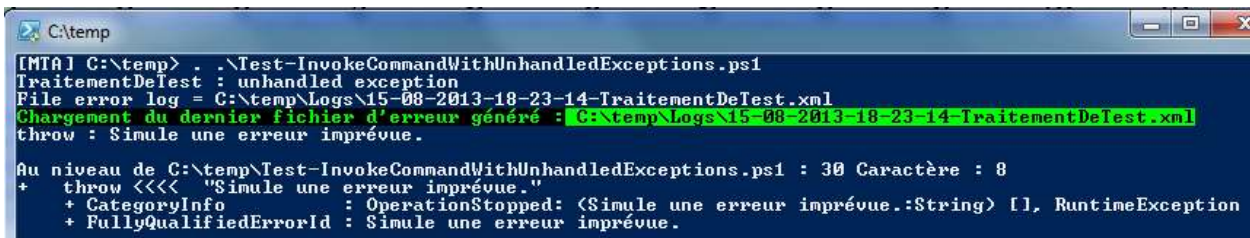
La fonction *Invoke-CommandWithUnhandledExceptions*, disponible dans les sources, propose une implémentation d'un tel mécanisme. Elle exécute du code dans un bloc try/catch et sérialise la collection d'erreur dans un fichier XML. Ce qui permet de la recharger ultérieurement dans une variable. Sa vocation n'est pas de libérer des ressources en cas d'erreur, mais d'enregistrer l'historique des erreurs pour un traitement donné.

La fonction *Get-LastError* se charge de cette opération, ce qui implique de créer pour chacun de vos traitements un répertoire de logs dédié.

Pour l'exemple suivant, les scripts de démonstration sont situés dans le répertoire *C:\Temp* :

```
#Modifier avec votre répertoire contenant les sources
Cd C:\Temp
. .\Test-InvokeCommandWithUnhandledExceptions.ps1
```

Le résultat d'exécution affichera ceci :



```
C:\temp
[MTA] C:\temp> . .\Test-InvokeCommandWithUnhandledExceptions.ps1
TraitementDeTest : unhandled exception
File error log = C:\temp\Logs\15-08-2013-18-23-14-TraitementDeTest.xml
Chargement du dernier fichier d'erreur généré : C:\temp\Logs\15-08-2013-18-23-14-TraitementDeTest.xml
throw : Simule une erreur imprévue.

Au niveau de C:\temp\Test-InvokeCommandWithUnhandledExceptions.ps1 : 30 Caractère : 8
+ throw <<<< "Simule une erreur imprévue."
+ CategoryInfo          : OperationStopped: (Simule une erreur imprévue.:String) [], RuntimeException
+ FullyQualifiedErrorId : Simule une erreur imprévue.
```

En lui précisant le chemin de log '*C:\Temp\Logs*' la fonction *Get-LastError* renvoie la dernière erreur générée par le script de test :

```

[MTA] C:\temp> $LastError=Get-LastError 'C:\temp\Logs'
Chargement du dernier fichier d'erreur généré : C:\temp\Logs\15-08-2013-18-23-14-TraitementDeTest.xml
[MTA] C:\temp> $LastError.InvocationInfo

MyCommand      :
BoundParameters : {}
UnboundArguments : {}
ScriptLineNumber : 30
OffsetInLine    : 8
HistoryId       : 16
ScriptName       : C:\temp\Test-InvokeCommandWithUnhandledExceptions.ps1
Line            : throw "Simule une erreur imprévue."
PositionMessage :

```

Notez que la fonction *Get-LastError* peut renvoyer une collection d'erreur, l'historique des erreurs donc. Le premier élément de la collection contiendra des informations sur l'erreur ayant arrêté le script.

8 Générer des classes d'exception

Powershell ne proposant pas d'instruction de création de classe, l'usage de l'instruction *throw* s'en trouve limité. Il reste possible d'utiliser certaines exceptions typées existantes ou d'en créer de nouvelles spécifiques à un traitement.

8.1 Type d'exception utilisable

Voici quelques exceptions pouvant être utilisées dans vos scripts :

System.ApplicationException	Cette exception est déclenchée lorsqu'une erreur fonctionnelle survient dans votre application ou votre script.
System.ArgumentException	Est déclenchée lorsqu'un des arguments d'une méthode ou une fonction est invalide.
System.NullReferenceException	Est déclenchée lorsqu'on adresse un objet ayant la valeur <i>null</i> .
System.NotImplementedException	Est déclenchée lorsqu'on appelle une méthode ou fonction qui n'est pas implémenté.
System.IndexOutOfRangeException	Est déclenchée lorsqu'on accède à un élément en dehors des limites d'un tableau.

Il en existe d'autres, mais leur usage sous Powershell apporterait une confusion. Par exemple, on peut être tenté d'utiliser la classe *System.OperationCanceledException*, mais celle-ci est dédiée à la gestion des threads...

8.2 Créer ses propres exceptions

Pour créer des exceptions spécifiques, il nous faut générer le code C# de déclaration de la classe puis le compiler à l'aide du cmdlet *Add-Type*.

On dérivera nos classes d'exception à partir de la classe *System.ApplicationException* et chaque classe définira les trois constructeurs suivant :

Nom	Description
Exception ()	Initialise une nouvelle instance de la classe Exception .

Exception (String)	Initialise une nouvelle instance de la classe Exception avec un message d'erreur spécifié..
Exception (String, Exception)	Initialise une nouvelle instance de la classe Exception avec un message d'erreur spécifié et une référence à l'exception interne qui est à l'origine de cette exception.

La règle de nommage préconisée est de postfixer le nom de la classe avec **Exception** :

```
#code C#
public class MonTraitementException : System.ApplicationException {...}
```

Vous trouverez dans les sources la fonction *New-ExceptionClass* dédiée à la création d'exception personnalisé.

L'exécution de l'instruction suivante :

```
New-ExceptionClass PsionicException,Posh4LogException -passthru
```

Génère uniquement du code C#:

```
[Serializable]
public class PsionicException : System.ApplicationException
{
    public PsionicException()
    {}
    public PsionicException(string message) : base(message)
    {}
    public PsionicException(string message, Exception innerException)
    : base(message, innerException)
    {}
}
[Serializable]
public class Posh4LogException : System.ApplicationException
{
    public Posh4LogException ()
    {}
    public Posh4LogException (string message) : base(message)
    {}
    public Posh4LogException (string message, Exception innerException)
    : base(message, innerException)
    {}
}
```

Celle-ci génère le code puis compile dynamiquement l'assembly

```
New-ExceptionClass PsionicException,Posh4LogException
$Exception=New-Object Posh4LogException "Appender inconnu."
$Exception|Select-Object *
Message      : Appender inconnu.
Data         : {}
InnerException :
TargetSite   :
StackTrace   :
HelpLink     :
Source       :
```

L'instruction suivante, utilise une hashtable déclarant des classes d'exception au sein d'un espace de nom :

```
$NameSpaces=@{
  'Module.Posh4Log'=@('GetAppenderException');
  'Module.Validation'=@('UserDataException','ADAttributException')
}
New-ExceptionClass $NameSpaces

$Message="Le champ MemberGroup doit être renseigné."
$Exception=New-Object Module.Validation.UserDataException $Message
```

Ces objets exceptions seront utilisés soit avec le cmdlet **Write-Error** soit avec l'instruction **Throw** :

```
Throw $Exception
```

9 Quelques opérations récurrentes

9.1 Tester si une erreur a eu lieu

La variable automatique **\$?** contient le résultat d'exécution de la dernière opération, à savoir si elle a réussi ou pas :

```
$ErrorActionPreference='Continue'
$Error.Clear()
xcopy.exe Inconnu.txt Nouveau.txt
Fichier introuvable - Inconnu.txt
0 fichier(s) copié(s)
if (-not $?)
{
  write-host "Dernière instruction en erreur."
  write-host "Code de sortie : $LastExitCode"
}
Dernière instruction en erreur.
Code de sortie : 4
```

9.2 Déterminer le retour d'exécution d'une commande

Quant à la variable automatique **\$LastExitCode** elle contient le code de sortie du dernier programme exécuté. Dans ce cas la collection d'erreur **\$Error** n'est pas renseignée, car le programme est exécuté en dehors du mécanisme d'erreur de Powershell.

Notez que la variable automatique **\$LastExitCode** n'existe pas au démarrage d'une session et qu'elle n'est modifiée que lors de l'appel d'un programme externe.

Autres exemples :

```
Cmd /c exit 0
"Exécution : $?"
```

```
"Code de sortie : $LastExitCode"
Excécution : True
Code de sortie : 0

Cmd /c exit 127
"Excécution : $?"
"Code de sortie : $LastExitCode"
Excécution : False
Code de sortie : 127

Dir
$?

True
$LASTEXITCODE
127 #Contient le dernier code retour d'exécution d'un programme externe
```

9.3 Redirection d'erreur

Pour prendre en compte les erreurs renvoyées par un programme externe, s'il les émet dans le flux standard *stderr*, on utilisera la redirection suivante :

```
$Error.Clear()
xcopy.exe Inconnu.txt Nouveau.txt 2>&1
0 fichier(s) copié(s)
xcopy.exe : Fichier introuvable - Inconnu.txt
```

Avec cette instruction Powershell se connecte au flux d'erreur du programme externe et redirige les erreurs de ce programme dans son flux d'erreur spécifique. On génère donc des erreurs Powershell tout en renseignant la collection *\$Error*.

Notez que dans ce cas le type est **RemoteException** :

```
$Error[0] | select *
PSMessageDetails :
Exception : System.Management.Automation.RemoteException: Fichier introuvable - Inconnu.txt
TargetObject : Fichier introuvable - Inconnu.txt
CategoryInfo : NotSpecified: (Fichier introuvable - Inconnu.txt:String) [], RemoteException
FullyQualifiedErrorId : NativeCommandError
ErrorDetails :
InvocationInfo : System.Management.Automation.InvocationInfo
PipelineIterationInfo : {0, 0}
```

Celle-ci redirige le flux d'erreur vers *\$null* :

```
xcopy.exe Inconnu.txt Nouveau.txt 2>$null
0 fichier(s) copié(s)
```

La collection *\$Error* est également renseignée, mais il n'y a plus d'affichage sur la console. Le paramétrage de *\$ErrorActionPreference* influencera ce comportement.

Cette construction est également possible :

```
$Error.Clear()
xcopy.exe Inconnu.txt Nouveau.txt 2> C:\Temp\Erreur.txt |
```

```
Foreach-object {write-host $_ -fore green}
0 fichier(s) copié(s)
Type C:\Temp\Erreur.txt
xcopy.exe : Fichier introuvable - Inconnu.txt

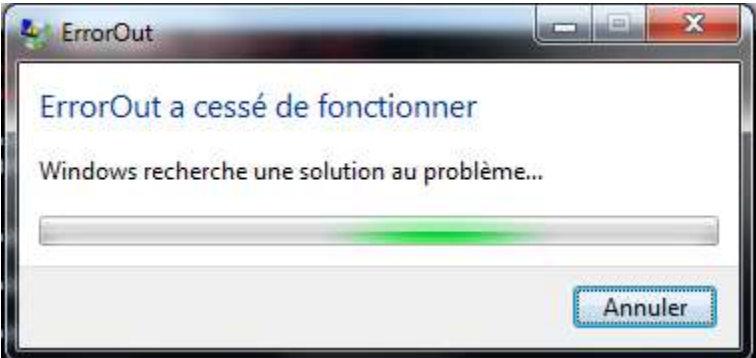
Au niveau de ligne : 1 Caractère : 10
+ xcopy.exe <<<< Inconnu.txt Nouveau.txt 2> C:\Temp\Erreur.txt|
+ CategoryInfo          : NotSpecified: (Fichier introuvable - Inconnu.txt:String) [], RemoteException
+ FullyQualifiedErrorId : NativeCommandError
```

Voir aussi le chapitre "Redirection Operators" de l'aide en ligne :

```
Get-Help about_operator
```

Enfin une application dotnet déclenchant une exception imprévue, ne renseignera pas la collection `$Error`. Le programme de test *ErrorOut.exe*, présent dans les sources, déclenche une telle exception :

```
$Error.Clear()
.\ErrorOut.exe
```



```
$Error.Count
0
```

Pour trapper cette exception sous Powershell on doit utiliser la redirection :

```
$Error.Clear()
.\ErrorOut.exe 2>&1
$Error.Count

.\ErrorOut.exe :
Au niveau de ligne : 1 Caractère : 15
+ .\ErrorOut.exe <<<< 2>&1
+ CategoryInfo          : NotSpecified: (:String) [], RemoteException
+ FullyQualifiedErrorId : NativeCommandError

Exception non gérée :
System.DivideByZeroException: Tentative de division par zéro.
à ErrorOut.ErrOut.Main()
$Error.Count
4
```

9.4 Flux standard

Les programmes de type console nommés *program.exe* et *program2.exe*, utilisent les flux standards du système *stderr* et *stdout* :

```
#code C#
Console.Error.WriteLine("Emit sur le flux d'erreur (stderr)");
Console.Out.WriteLine("Emit sur le flux de sortie (stdout)");
```

Résultat d'exécution :

```
cd "VotreRépertoireDeDemo"
$error.Clear()
.\program.exe 1 > "C:\Temp\Erreurs.txt"
Emit sur le flux d'erreur (stderr)
Tentative de division par zéro.
type "C:\Temp\Erreurs.txt"
Emit sur le flux de sortie (stdout)
$error.Count #n'est pas renseignée
0
.\program.exe 2>&1
Emit sur le flux de sortie (stdout)
Au niveau de ligne : 1 Caractère : 14
+ .\program.exe <<<< 2>&1
+ CategoryInfo          : NotSpecified: (Emit sur le flux d'erreur (stderr):String) [], RemoteException
+ FullyQualifiedErrorId : NativeCommandError

Tentative de division par zéro.
$error.Count
2
```

Program.exe utilise la méthode `WriteLine`, *program2.exe* la méthode `Write`. L'usage de l'une ou de l'autre influence le nombre d'erreur récupéré par le mécanisme de redirection des erreurs sous Powershell.

Vous pouvez également consultez ce poste :

http://powershell-scripting.com/index.php?option=com_joomlaboard&Itemid=76&func=view&id=12899&catid=14

9.4.1 Les nouveaux flux sous Powershell version 3

On peut désormais rediriger la plupart des flux :

- * *All output*
- 1 *Success output*
- 2 *Errors*
- 3 *Warning messages*
- 4 *Verbose output*
- 5 *Debug messages*

```
Do-Something 3> warning.txt # writes warning output to warning.txt
Do-Something 4>> verbose.txt # Appends verbose.txt with the verbose output
Do-Something 5>&1 # writes debug output to the output stream
```

```
Do-Something *> out.txt # Redirects all streams (output, error, warning, verbose, and debug) to out.txt
```

Pour le détail consultez l'aide en ligne :

```
Get-Help About_Redirection.txt
```

Voir également dans les sources le script d'exemple *Test-Redirection.ps1*.

9.5 Propriété de formatage

Il existe une propriété nommée *WriteErrorStream*, lié au formatage d'un résultat d'instruction,

Vous trouverez le détail dans le post suivant :

<http://www.beefycode.com/post/Formatting-Individual-PowerShell-Outputs-as-Errors.aspx>

10 Conclusion

Au travers de ces pages vous avez pu constater que la gestion d'erreur est un sujet relativement simple, mais que son implémentation sous Powershell nécessite de connaître les nombreux comportements et les nombreuses possibilités offertes. N'hésitez pas à consulter régulièrement le site MSConnect, car vous pourriez être amené à contourner des bugs du runtime Powershell.

Faute de temps, je n'ai pas traité la gestion des erreurs liées aux jobs (remoting), un prochain chapitre peut-être.

Les tests étant une étape préliminaire à la gestion d'erreur et le debug de script l'étape suivante, vous trouverez de nombreuses ressources sur le net pour aborder ces sujets.

Enfin souvenez-vous que nous avons le droit de faire des erreurs, mais que nous devons tout faire pour ne pas les reproduire.