

Les variables contraintes sous PowerShell

Par Laurent Dardenne, le 16 mai 2008.



Niveau

Débutant	Avancé	Confirmé
	<input type="checkbox"/>	

Le SDK de PowerShell contient quelques classes pouvant être utilisées directement dans des scripts, il en est une que Bruce Payette a mentionné dans un document contenant les corrections de son ouvrage «Windows PowerShell in action». Celle-ci autorise la création de variables contraintes intégrant des règles de validation, je vous propose dans ce tutoriel de détailler leurs manipulation.

Chapitres

1	RAPPELS SUR LES VARIABLES	3
2	LES VARIABLES CONTRAINTES	5
2.1	AJOUTER UN ATTRIBUT	5
2.2	TESTER L'EXISTENCE D'UN ATTRIBUT	6
2.3	MODIFIER UN ATTRIBUT	6
2.4	SUPPRIMER UN ATTRIBUT	7
2.5	COPIER UN ATTRIBUT	7
2.6	TESTER LES CONTRAINTES POUR UNE VALEUR DONNEE	7
3	LES ATTRIBUTS DISPONIBLES	8
3.1	VALIDATENOTNULLOREMPTYATTRIBUTE.....	9
3.2	VALIDATERANGEATTRIBUTE.....	9
3.3	VALIDATECOUNTATTRIBUTE.....	9
3.4	VALIDATELENGTHATTRIBUTE.....	10
3.5	VALIDATESETATTRIBUTE	11
3.6	VALIDATEPATTERNATTRIBUTE	11
3.7	LES ATTRIBUTS ALLOWXXX	11
3.8	LA COMBINAISON D'ATTRIBUTS	12
3.9	RETROUVER LE DETAIL D'UNE EXCEPTION DE VALIDATION	12
4	PASSER UNE VARIABLE CONTRAINTE EN PARAMETRE	14
4.1	GESTION DES EXCEPTIONS.....	15
5	DEVELOPPER CES PROPRES CONTRAINTES	16

Pré-requis

Pour la première partie du tutoriel une aisance avec les principes et concepts de script de PowerShell, pour la seconde partie des notions de bases sur la programmation .NET en C#.

Télécharger les scripts : <ftp://ftp-developpez.com/laurent-dardenne/articles/Windows/PowerShell/VariablesContraintes/fichiers/PSVariablesContraintes.zip>

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

1 Rappels sur les variables

Dans un script PowerShell les variables n'ont pas besoin d'être déclarées avant leur utilisation.

Créons une variable contenant un entier :

```
$V=10
$V|Gm
  TypeName: System.Int32
Name      MemberType Definition
----      -
...

```

Elles sont temporairement enregistrées dans le provider *variable* : ([http://msdn2.microsoft.com/en-us/library/system.management.automation.provider.containercmdletprovider\(VS.85\).aspx](http://msdn2.microsoft.com/en-us/library/system.management.automation.provider.containercmdletprovider(VS.85).aspx)) :

```
PS C:\TempP> cd variable:
PS variable:\> dir v
Name      Value
----      -
V         10

```

Ici on affiche l'objet contenant la variable V, il est possible de récupérer la définition d'une variable à l'aide du cmdlet **Get-Variable** :

```
$DefV=Get-Variable v
$DefV
Name      Value
----      -
V         10

```

Affichons les membres que cet objet variable contient :

```
$DefV|gm
  TypeName: System.Management.Automation.PSVariable
Name      MemberType Definition
----      -
...
IsValidValue Method System.Boolean IsValidValue(Object value)
Attributes   Property System.Collections.ObjectModel.Collection`1[System.Attribute,
Description  Property System.String Description {get;set;}
Name         Property System.String Name {get;}

```

Options	Property	System.Management.Automation.ScopedItemOptions	Options {get;set;}
Value	Property	System.Object	Value {get;set;}
...			

On récupère la définition dans une instance de la classe PSVariable, classe provenant du sdk de PowerShell, ([http://msdn2.microsoft.com/en-us/library/system.management.automation.psvariable_members\(VS.85\).aspx](http://msdn2.microsoft.com/en-us/library/system.management.automation.psvariable_members(VS.85).aspx)).

Nous avons vu que les propriétés *value* et *name* référencent bien la variable \$V, la propriété *description* permet de commenter l'usage de la variable, voyons l'usage de la propriété *Options*.

Elle définit la façon dont la donnée est employée dans la session. C'est une énumération de type

ScopedItemOptions ([http://msdn2.microsoft.com/en-us/library/system.management.automation.scopeditemoptions\(vs.85\).aspx](http://msdn2.microsoft.com/en-us/library/system.management.automation.scopeditemoptions(vs.85).aspx))

Nom de membre	Description
AllScope	La donnée est copiée dans toutes les nouvelles portées qui sont créées.
Constant	La donnée ne peut être ni supprimée ni modifiée.
None	Pas d'options.
Private	La donnée est vue seulement dans la portée courante (introuvable des portées enfant lors d'une consultation).
ReadOnly	La donnée peut être supprimée mais pas modifiée.

```
$DefV.Options=[System.Management.Automation.ScopedItemOptions]"Allscope"
```

On retrouve cette propriété comme paramètres du cmdlet **New-Variable** :

```
PS Variable:\> New-Variable V2 -option Constant -value 10
PS Variable:\> $V2
10
PS Variable:\> $V2=9
```

Impossible de remplacer la variable V2, car elle est constante ou en lecture seule.

```
$DefV.Options=[System.Management.Automation.ScopedItemOptions]"Constant"
```

Exception lors de la définition de « Options » : « La variable existante V ne peut pas être définie comme constante. Les variables peuvent uniquement être définies comme constantes au moment de la création. »

La valeur *Constant* ne peut être utilisée qu'avec le cmdlet **New-Variable**. Une fois déclaré *Constant* ou *ReadOnly*, la modification de la propriété *Options* d'une variable n'est plus possible :

```
$DefV.Options=[System.Management.Automation.ScopedItemOptions]"Allscope,ReadOnly"
$DefV.Options
ReadOnly, AllScope
$V=15
```

Impossible de remplacer la variable V, car elle est constante ou en lecture seule.

```
$DefV.Options=[System.Management.Automation.ScopedItemOptions]"Allscope"
```

Exception lors de la définition de « Options » : « Impossible de remplacer la variable V, car elle est constante ou en lecture seule. »

Il nous reste à voir dans les propriétés celle nommée *Attributes*. C'est elle qui contient les attributs optionnels de validation.

2 Les variables contraintes

L'objectif des contraintes est de valider le contenu d'une variable en autorisant ou non certaines valeurs et ce à l'aide de règles portées par des attributs

(<http://www.msdnacademie.be/DocsCenter/DL.aspx?Doc=73>).

A l'origine l'usage des attributs de validation est de contrôler les paramètres d'un cmdlet codé dans un langage dotNET.

Un exemple d'utilisation en C# :

```
[Parameter(
    Position = 0,
    ValueFromPipeline = true,
    ValueFromPipelineByPropertyName = true
)]
[ValidateNotNullOrEmpty] # Attribut de validation
public string[] Name
{
    get { return processNames; }
    set { processNames = value; }
}
private string[] processNames;
```

Voyons les manipulations de base autour de cette classe en déclarant l'attribut *ValidateNotNullAttribute* sur une variable contenant un entier.

2.1 Ajouter un attribut

Par défaut la liste des attributs contenue dans la propriété nommée *Attributes* est vide. Pour ajouter un attribut il nous faut d'abord le créer puisque dans ce contexte nous utilisons les classes d'attributs .NET qui ne sont pas pris en charge nativement par PowerShell v1.0 :

```
(Get-Variable V).Attributes
$Attribut=New-object System.Management.Automation.ValidateNotNullAttribute
$DefV.Attributes.add($Attribut)
```

Vérifions le nouveau contenu de la collection d'attributs :

```
$DefV.Attributes
TypeId
-----
System.Management.Automation.ValidateNotNullAttribute
```

Affectons la valeur *\$Null* à notre variable contrainte :

```
$V=$Null
```

Validation impossible en raison d'une valeur non valide () pour la variable V.

Cette valeur ne respectant pas la règle fixée par la contrainte on ne peut plus désormais affecter la valeur *\$Null* à cette variable et uniquement celle-ci.

Les exceptions déclenchées en cas de violation de contrainte sont de la classe **ValidationMetadataException**

([http://msdn2.microsoft.com/en-us/library/system.management.automation.validationmetadataexception\(VS.85\).aspx](http://msdn2.microsoft.com/en-us/library/system.management.automation.validationmetadataexception(VS.85).aspx)), déclarée dans l'espace de nom System.Management.Automation ([http://msdn2.microsoft.com/en-us/library/system.management.automation\(VS.85\).aspx](http://msdn2.microsoft.com/en-us/library/system.management.automation(VS.85).aspx)).

Pour alléger la suite du texte je ne référencerai plus explicitement cet espace de nom, de plus j'utiliserai dans le script le raccourci d'écriture suivant :

```
$SMA="System.Management.Automation"  
$Attribut=New-object "$SMA.ValidateNotNullAttribute"
```

Il est tout à fait possible d'ajouter un second attribut de la même classe mais je n'en vois pas l'intérêt pour l'attribut **ValidateNotNullAttribute**.

Note : L'usage de **Clear-Variable V** pour supprimer cette variable contrainte provoquera la même exception que l'affectation de la valeur *\$Null*. On utilisera donc **Remove-Variable NomdeVariable**, sauf pour celles qui sont définies comme constantes ou celles qui sont la propriété du système.

2.2 Tester l'existence d'un attribut

On utilise les méthodes de la collection générique ([http://msdn2.microsoft.com/en-us/library/ms132397\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms132397(VS.80).aspx)) qui propose la méthode *Contains*. On lui passe en paramètre une nouvelle instance de l'attribut recherché :

```
$DefV.Attributes.Contains( (New-object "$SMA.ValidateNotNullAttribute") )
```

Ou si vous souhaitez récupérer l'index du premier attribut existant :

```
$DefV.Attributes.IndexOf((New-object "$SMA.ValidateNotNullAttribute"))
```

L'appel de la méthode *IndexOf* renvoie -1 en cas d'échec.

2.3 Modifier un attribut

Il n'est pas possible de modifier un attribut à moins de récupérer ses valeurs, de le supprimer puis de le recréer avec de nouvelles valeurs, tout en s'assurant que l'ancien contenu reste valide :

```
$V=10  
$DefV=Get-Variable v  
$ContraintedEtendu=New-object "$SMA.ValidateRangeAttribute" 2,15  
$DefV.Attributes.add($ContraintedEtendu)  
$I=$DefV.Attributes.IndexOf((New-object "$SMA.ValidateRangeAttribute"  
2,15))  
$DefV.Attributes[$I].MinRange=5
```

« *MinRange* » est une propriété en lecture seule.

2.4 Supprimer un attribut

Pour supprimer le premier attribut identique trouvé on utilisera la méthode *Remove* :

```
$DefV.Attributes.Remove((new-object "$SMA.ValidateNotNullAttribute"))
```

Ou *RemoveAt* si on connaît l'index :

```
$DefV.Attributes.RemoveAt(NumeroIndex)
```

2.5 Copier un attribut

```
Copy-Item -Path variable:v -Destination variable:v2  
Get-Variable v|fl  
gv v2|fl
```

Pour le moment le cmdlet *Copy-Item*, ne recopie pas les contraintes et les options déclarées, je pense que nous sommes ici en présence d'une copie incomplète ("shallow copy") alors qu'une copie complète est nécessaire ("deep copy"). Utilisons la sérialisation d'objet pour régler provisoirement ce problème :

```
gv v|Export-Clixml variable-v.xml  
$Nouvelle=Import-Clixml variable-v.xml  
#On doit modifier le nom enregistré  
$Nouvelle.name= "Nouvelle"  
$v|gm  
$Nouvelle|gm
```

Il reste un léger soucis, avec cette approche la variable *\$Nouvelle* est de type **PsVariable** et non pas **Int32**. Pour le moment je n'ai pas trouvé de solution à moins de développer un cmdlet *Clone-Variable...*

2.6 Tester les contraintes pour une valeur donnée

La méthode *IsValidValue* vérifie si la valeur passée en paramètre respecte les contraintes existantes :

```
$DefV.IsValidValue($Null)  
False  
$DefV.IsValidValue(10)  
True
```

Note : Il n'est pas possible de tester une valeur contrainte par contrainte.

3 Les attributs disponibles

Voici la liste des attributs que l'on peut déclarer pour une variable :

Constructeur de l'attribut	Usage et type de donnée ciblée
ValidateNotNullAttribute()	La valeur de la variable ne peut être égale à <i>\$Null</i> <i>Tous</i>
ValidateNotNullOrEmptyAttribute()	La valeur ne peut être égale à <i>\$Null</i> ou vide. <i>String, Collection</i> <u>Les collections doivent implémentées une de ces interfaces</u> : <i>IList, ICollection, IEnumerable, IEnumerator</i> .
ValidateRangeAttribute (<i>Object</i> minRange, <i>Object</i> maxRange)	Les valeurs autorisées sont celles d'une étendue. Par exemple entre 2 et 8. <i>Scalaire</i> La classe de l'objet doit implémenter l'interface <i>IComparable</i> . Le type de l'élément doit être de même type que la propriété <i>minRange</i> .
ValidateCountAttribute (<i>int</i> minLength, <i>int</i> maxLength)	Nombre d'éléments autorisés dans une collection. <i>String, tableau uniquement ?</i>
ValidateLengthAttribute (<i>int</i> minLength, <i>int</i> maxLength)	La taille autorisée doit être comprise entre une valeur minimum et une valeur maximum. <i>String</i>
ValidateSetAttribute (params <i>string</i> [] validValues)	Les valeurs autorisées sont celles contenues dans une énumération. <i>String ou tous après conversion via la méthode ToString()</i> . Note : une seule valeur équivaut à l'option <i>Constant</i> .
ValidatePatternAttribute (<i>string</i> regexPattern)	Les valeurs autorisées sont celles correspondantes à une expression régulière unique. <i>String ou tous après conversion via la méthode ToString()</i> .

3.1 *ValidateNotNullOrEmptyAttribute*

Contraint une variable avec un attribut null ou vide :

```
$ChaineContrainte="Test"  
$ContrainteNullouVide=New-object "$SMA.ValidateNotNullOrEmptyAttribute"  
(Get-Variable ChaineContrainte).Attributes.add($ContrainteNullouVide)
```

```
$ChaineContrainte=""
```

Validation impossible en raison d'une valeur non valide () pour la variable ChaineContrainte.

```
$ChaineContrainte=$Null
```

Validation impossible en raison d'une valeur non valide () pour la variable ChaineContrainte.

3.2 *ValidateRangeAttribute*

Contraint une variable avec un attribut d'étendue :

```
$VariableContrainte=5  
$ContraintedEtendu=New-object "$SMA.ValidateRangeAttribute" 2,8  
(Get-Variable VariableContrainte).Attributes.add($ContraintedEtendu)
```

```
$VariableContrainte=2
```

```
$VariableContrainte=9
```

Validation impossible en raison d'une valeur non valide (9) pour la variable VariableContrainte.

3.3 *ValidateCountAttribute*

Contraint une variable, ici de type tableaux, avec un attribut précisant un nombre d'élément :

```
$TableauContraint=@(1,2)  
$ContrainteNbElement=New-object "$SMA.ValidateCountAttribute" 1,3  
(Get-Variable TableauContraint).Attributes.add($ContrainteNbElement)  
$TableauContraint+=3  
$TableauContraint+=4
```

Validation impossible en raison d'une valeur non valide (System.Object[]) pour la variable TableauContraint.

```
(Get-Variable TableauContraint).Attributes|fl *
```

```
MinLength : 1
```

```
MaxLength : 3
```

```
TypeId : System.Management.Automation.ValidateCountAttribute
```

Ici la valeur de la propriété *MinLength* de notre attribut interdit un tableau vide ou null :

```
$TableauContraint=@()
```

Validation impossible en raison d'une valeur non valide (System.Object[]) pour la variable TableauContraint.

```
$TableauContraint=$Null
```

Validation impossible en raison d'une valeur non valide (System.Object[]) pour la variable TableauContraint.

Pour éviter ceci vous devrez affecter 0 à la propriété *MinLength*.

Note :

Cet attribut semblait utilisable avec une collection de type *ArrayList* mais l'ajout de l'attribut provoque une exception :

```
$TableauContraint=New-Object System.Collections.ArrayList
$TableauContraint.add(1)
```

```
$ContrainteNbElement=New-object "$SMA.ValidateCountAttribute" 0,3
(Get-VARIABLE TableauContraint).Attributes.add($ContrainteNbElement)
```

Exception lors de l'appel de « Add » avec « 1 » argument(s) : « Impossible d'ajouter l'attribut, car la variable TableauContraint ayant la valeur 1 2 ne serait plus valide. »

Voir le chapitre *Retrouver le détail d'une exception de validation*.

3.4 *ValidateLengthAttribute*

Contraint une variable, ici de type chaîne de caractères, avec un attribut précisant la taille minimum et maximum à respecter :

```
$ChaineContrainteLongueur="12345"
$ContrainteLongueur=New-object "$SMA.ValidateLengthAttribute" 2,6
(Get-VARIABLE ChaineContrainteLongueur).Attributes.add($ContrainteLongueur)
$ChaineContrainteLongueur="1234567"
```

Validation impossible en raison d'une valeur non valide (1234567) pour la variable ChaineContrainteLongueur.

```
$ChaineContrainteLongueur="1"
```

Validation impossible en raison d'une valeur non valide (1) pour la variable ChaineContrainteLongueur.

Attention à ne pas confondre cet attribut avec *ValidateCountAttribute* :

```
[String[]] $TableauContraint=@("12","12345")
$ContrainteLongueur=New-object "$SMA.ValidateLengthAttribute" 2,6
(Get-VARIABLE TableauContraint).Attributes.add($ContrainteLongueur)
1..10|% {$TableauContraint+="test"}
$TableauContraint+="1234567"
$TableauContraint+="1"
```

Avec cette contrainte on peut insérer dans le tableau de chaîne autant d'élément que l'on veut mais chaque chaîne doit respecter la contrainte de longueur.

Et dans cet exemple :

```
$ChaineContrainte="12"
$ContrainteNbElement=New-object "$SMA.ValidateCountAttribute" 1,3
(Get-VARIABLE $ChaineContrainte).Attributes.add($ContrainteNbElement)
$ChaineContrainte="1234"
$ChaineContrainte="1"
```

Puisque le type **String** est représenté en interne par un tableau de caractère, cet attribut est ici équivalent à *ValidateLengthAttribute*.

3.5 *ValidateSetAttribute*

Contraint une variable, ici de type chaîne de caractères, avec un attribut précisant les seules valeurs autorisées :

```
$Ensemble="Un"
$ContrainteEnsemble=New-object "$SMA.ValidateSetAttribute" ("Un","Deux","Trois")
(Get-Variable Ensemble).Attributes.add($ContrainteEnsemble)
$Ensemble="deux"      # Insensible à la casse
$Ensemble="six"
Validation impossible en raison d'une valeur non valide (six) pour la variable Ensemble.
$Ensemble=$Null
```

Validation impossible en raison d'une valeur non valide () pour la variable Ensemble.

Notez que la valeur initiale de la variable doit être contenue dans l'ensemble. Ici aussi on utilisera le cmdlet **Remove-Variable** pour supprimer la variable *\$Ensemble*.

Cette classe propose un champ nommé *IgnoreCase*, sa valeur par défaut est **true**.

3.6 *ValidatePatternAttribute*

Contraint une variable, ici de type chaîne de caractères, avec un attribut précisant les seules valeurs autorisées correspondantes à une expression régulière unique :

```
$Chaine="8AB5"
$ContraintePattern=New-object "$SMA.ValidatePatternAttribute" "[0-9]AB[5-7]"
(Get-Variable Chaine).Attributes.add($ContraintePattern)
$Chaine="Deux"
Validation impossible en raison d'une valeur non valide (Deux) pour la variable Chaine.
$Chaine=$Null
```

Validation impossible en raison d'une valeur non valide () pour la variable Chaine.

Cet attribut peut servir par exemple à valider une adresse IP :

```
"^(25[0-5]|2[0-4]\d|[0-1]?\d?\d)(\.(25[0-5]|2[0-4]\d|[0-1]?\d?\d)){3}$"
```

Cette classe propose un champ nommé *options* spécifiant un comportement particulier pour l'expression régulière, par exemple *IgnoreCase*.

3.7 *Les attributs Allowxxx*

Dans l'exemple précédent on aurait pu penser que les attributs suivants *AllowEmptyCollectionAttribute*, *AllowEmptyStringAttribute* et *AllowNullAttribute* pouvaient nous aider à autoriser une chaîne *\$Null* ou une chaîne respectant l'expression régulière :

```
$Chaine="8AB5"
$ContrainteNullAutorise=New-object "$SMA.AllowNullAttribute"
```

```
(Get-Variable Chaine).Attributes.add($ContrainteNullAutorise)
$ContraintePattern=New-object "$SMA.ValidatePatternAttribute" "[0-9]AB[5-7]"
(Get-Variable Chaine).Attributes.add($ContraintePattern)
$Chaine=$Null
```

Validation impossible en raison d'une valeur non valide () pour la variable Chaine.

Malheureusement cette combinaison n'est pas possible puisque le nouveau contenu doit être valide pour toutes les contraintes déclarées.

3.8 La combinaison d'attributs

Il reste toutefois possible de combiner des attributs si la condition est un **et** et pas un **ou**.

Contraindre un tableau sur le nombre d'élément **et** sur la valeur de chaque élément :

```
$TableauContraint=@(2,4)
$ContrainteNbElement=New-object "$SMA.ValidateCountAttribute" 1,3
(Get-Variable TableauContraint).Attributes.add($ContrainteNbElement)
$ContraintedEtendu=New-object "$SMA.ValidateRangeAttribute" 2,7
(Get-Variable TableauContraint).Attributes.add($ContraintedEtendu)
$TableauContraint+=8
```

Validation impossible en raison d'une valeur non valide (System.Object[]) pour la variable TableauContraint.

```
$TableauContraint+=5
```

```
$TableauContraint+=5
```

Validation impossible en raison d'une valeur non valide (System.Object[]) pour la variable TableauContraint.

La première exception est déclenchée par la contrainte d'étendue et la seconde par la contrainte de nombre d'élément.

3.9 Retrouver le détail d'une exception de validation

Comme l'attribut *ValidateRangeAttribute* attend deux paramètres de type objet on peut valider une étendue de date :

```
[DateTime] $Date1="01/01/1900"
[DateTime] $Date2="12/31/1999"
[DateTime] $Date="10/09/1950"

$ContraintedEtendu=New-object "$SMA.ValidateRangeAttribute" $Date1, $Date2
(Get-Variable Date).Attributes.add($ContraintedEtendu)
[DateTime] $Date="01/01/2000"
```

Validation impossible en raison d'une valeur non valide (01/01/2000 00:00:00) pour la variable Date.

En affichant le détail de l'exception, à l'aide de ce script

(<http://blogs.msdn.com/powershell/archive/2006/12/07/resolve-error.aspx>) on ne peut pas obtenir la véritable raison de cette exception. A savoir si c'est la borne inférieure ou supérieure qui n'est pas respectée.

Comme dans ce cas l'exception est de type *ValidationMetadataException* on peut essayer d'afficher le détail de cet appel à l'aide du Cmdlet **Trace-Command**.

Il nécessite un paramètre nommé *-name* voici ce que nous dit la documentation à son propos :

Détermine quels composants Windows PowerShell sont tracés. Entrez le nom de la source de trace de chaque composant. Les caractères génériques sont autorisés. Pour rechercher les sources de trace sur votre ordinateur, tapez " Get-TraceSource ".

En utilisant le cmdlet cité on peut tenter de retrouver un listener traçant les appels sur les métadonnées :

```
Get-TraceSource| ? {$_.description -match "meta"}
Name      : Metadata
Description : Metadata

Name      : CommandMetadata
Description : The metadata associated with a bindable object in MSH.

Name      : ParameterMetadat
Description : The metadata associated with a bindable object type in MSH.

Name      : CompiledCommandP
Description : The metadata associated with a parameter that is attached to a bindable object in MSH.

Name      : CompiledCommandA
Description : The metadata associated with an attribute that is attached to a bindable object in MSH.

Name      : ParameterSetSpec
Description : The metadata associated with a parameterset in a bindable object in MSH.
```

(MSH pour MonadShell)

Essayons avec celui nommé *Metadata* :

```
Trace-command -Name Metadata -Option All -Exp {[DateTime]
$date="01/01/2000"} -FilePath Trace.log
select-string -Path trace.log -Pattern ": Exception"|% {$_.Line}
Metadata Information: 0 : Exception      System.Management.Automation.ValidationMetadataException:
L'argument (01/01/2000 00:00:00) était supérieur à la plage maximale (31/12/1999 00:00:00).

Metadata Information: 0 : Exception      System.Management.Automation.ValidationMetadataException:
Validation impossible en raison d'une valeur non valide (01/01/2000 00:00:00) pour la variable Date.
```

Ainsi on récupère l'exception d'origine qui n'est malheureusement pas placée dans la propriété ***\$Error[0].Exception.InnerException***.

En consultant le fichier on obtient une information supplémentaire à savoir que c'est la méthode *ValidateRangeAttribute.ValidateElement()* qui provoque l'exception d'origine. On obtient ainsi la cause réelle de l'exception et pas un message générique.

4 Passer une variable contrainte en paramètre

Il reste un point à vérifier qui est comment passer une variable contrainte à une fonction. Reprenons un des précédents exemples :

```
$Ensemble="Un"
$ContrainteEnsemble=New-object "$SMA.ValidateSetAttribute" ("Un","Deux","Trois")
(Get-Variable Ensemble).Attributes.add($ContrainteEnsemble)
$Ensemble="Deux"
```

Déclarons une fonction comportant un paramètre :

```
function Test($A){
    write-host $A
    $A="Quatre" # Modification notre variable passée en paramètre
    write-host $A}
Test $Ensemble
Deux
Quatre
```

Dans ce cas comme on passe une copie de la valeur contenue dans notre variable, les contraintes n'existent plus.

La manipulation de notre variable au travers du spécificateur de portée *global*: reste possible. Essayons en modifiant le type de notre paramètre :

```
function Test([System.Management.Automation.PSVariable] $A){
    write-host $A
    $A.Value="Quatre"
    write-host $A}
Test (gv ensemble)
Exception lors de la définition de « Value » : « Validation impossible en raison d'une valeur non valide (Quatre) pour la variable Ensemble. »
```

Avec cette manière de procéder les contraintes sont bien présentes, on doit toutefois utiliser une indirection en manipulant la propriété *Value*. Notez que le type de l'exception diffère.

PowerShell propose un type de donnée [REF] qui permet de passer une variable par adresse, on manipule donc un pointeur sur un objet :

```
function Test([REF] $A){
    write-host $A.Value
    $A.Value="Quatre"
    write-host $A.Value}
```

On peut déclarer une référence (un pointeur) de la manière suivante :

```
$Pointeur=[REF] $Ensemble
Test $Pointeur
```

Exception lors de la définition de « Value » : « Validation impossible en raison d'une valeur non valide (Quatre) pour la variable Ensemble. »

Ou en en simplifiant :

```
Test ([REF] $Ensemble)
```

Exception lors de la définition de « Value » : « Validation impossible en raison d'une valeur non valide (Quatre) pour la variable Ensemble. »

Ici on contraint une variable d'un type particulier mais on ne crée pas un type de variable.

Cette manière de procéder limite, à mon avis, l'usage des variables contraintes bien qu'elles restent intéressantes dans certains contextes.

Voir : ScriptCmdlet de la version 2 de PowerShell (<http://technet.microsoft.com/en-us/library/bb978611.aspx>)

4.1 Gestion des exceptions

Si, en cas d'erreur d'affectation, on souhaite connaître le nom de la variable incriminée on doit analyser le message d'erreur car le type de l'exception est unique pour toutes les variables contraintes :

```
Function TestExceptions{
    trap [System.Management.Automation.ValidationMetadataException]
    { #Récupère trois groupes de chaîne de caractères
      #dans la variable automatique $Matches
      if ($_ -match "^(.*) valide \((.*)\) pour la variable (.*)$" )
      { #Le nom de la variable
        $V=$Matches[3]
        write-Host $V" " -for yellow -noNewLine
        #Le contenu actuel
        write-Host (invoke-expression "`$V")" " -noNewLine
        #Le contenu erroné
        write-Host $Matches[2] -for red -noNewLine
        #Traitement optionel d'après le nom de la variable
        switch ($V){
            "TableauContraint" { write-Host " Switch : variable $V"}
        }
        Continue
      }
    }
}

$TableauContraint=@(2,4)
$ContrainteNbElement=New-object "$SMA.ValidateCountAttribute" 1,3
(Get-Variable TableauContraint).Attributes.add($ContrainteNbElement)
$ContraintedEtendu=New-object "$SMA.ValidateRangeAttribute" 2,7
(gv TableauContraint).Attributes.add($ContraintedEtendu)
$TableauContraint=8
```

```

[DateTime] $Date1="01/01/1900"
[DateTime] $Date2="12/31/1999"
[DateTime] $Date="10/09/1950"

$ConstrainedEtendu=New-object "$SMA.ValidateRangeAttribute" $Date1, $Date2
(gv Date).Attributes.add($ConstrainedEtendu)
[DateTime] $Date="01/01/2000"
}

TestExceptions

```

Soyez attentif aux environnements multilingues.

5 Développer ces propres contraintes

Pour ce faire on dérive notre nouvelle classe de *ValidateArgumentsAttribute*.

Nous allons contraindre une variable de type string à contenir un nom de fichier existant, nous nommerons notre classe *ValidatePathAttribute*. Ici le terme path référence le chemin d'un système de fichier et pas la notion de path d'un provider (cf. **Test-Path**).

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Management.Automation;
using System.IO;

namespace PowerShell.Attributes
{
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
    public sealed class ValidatePathAttribute : ValidateArgumentsAttribute
    {
        protected override void Validate(object arguments, EngineIntrinsics engineIntrinsics)
        {
            string FileName = arguments as string;
            if (string.IsNullOrEmpty(FileName))
            {
                throw new ValidationMetadataException("Le nom de fichier est null ou vide.");
            }
        }
    }
}

```

```

FileInfo fileInfo = new FileInfo(fileName);

if (!fileInfo.Exists)
{
    throw new ValidationMetadataException(String.Format("Le nom de fichier n'existe pas : {0}", fileName));
}
}
}
}
}

```

A noter que la variable *engineIntrinsics* permet d'accéder aux informations de la session PowerShell courante, celle où est déclarée notre variable à valider.

Sous Powershell on utilisera cet assembly ainsi :

```

# Assembly avec un nom fort
#[void][Reflection.Assembly]::LoadWithPartialName("PowerShell.Attributes")

# Assembly sans nom fort, on doit préciser le chemin complet
$FullPath =
"C:\PS\Test\ValidatePathAttribute\ValidatePathAttribute\bin\Debug\Validate
PathAttribute.dll"

[void][Reflection.Assembly]::LoadFile($FullPath)
$File="C:\Boot.ini" # Existe tjr sous windows XP
$DefFile =Get-Variable File
$Attribut=New-object "PowerShell.Attributes.ValidatePathAttribute"
$DefFile.Attributes.add($Attribut)
$File="X:\fichierInconnu.txt"

```

Validation impossible en raison d'une valeur non valide (X:\fichierInconnu.txt) pour la variable File.