PowerShell version 2 :
Les fonctions avancées.

Par Laurent Dardenne, le 13 mars 2010.

Niveau

Débutant	Avancé	Confimé

La version 2 de PowerShell apporte de nombreuses évolutions, l'une concerne l'écriture de cmdlet en code natif à l'aide de fonctions dites avancées utilisant différents types d'attributs. Dans le texte suivant je vous propose d'étudier la création de cmdlets sans utiliser de langage compilé dotnet.

Une bonne connaissance des principes de base de PowerShell, notamment celui du pipeline, du fonctionnement des cmdlets et des classes dotnet, facilitera la lecture de ce tutoriel qui est tout de même orienté développeur. Difficile de faire autrement sur un tel sujet, mais cela reste un tutoriel sur le scripting avancé sous PowerShell.

Merci à Thomas Garcia pour sa relecture et ses corrections orthographiques.

Les fichiers sources :

ftp://ftp-developpez.com/laurent-dardenne/articles/Windows/PowerShell/Les-fonctions-et-scripts-avancees-sous-PowerShell-version-2/fichiers/Les-fonctions-et-scripts-avancees-sous-PowerShell-version-2.zip

Testé avec PowerShell V2 sous Windows XP sp3.

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

Chapitres

1 Quelles sont les avancées ?

Les fonctions et scripts dits avancés sont des fonctions ou des scripts autorisant l'usage d'attributs comme on en utilise dans les déclarations de cmdlets codés avec un langage dotnet compilé. Ainsi, on peut nativement placer des règles de validation et des contraintes sur tout ou partie des paramètres. Nous verrons également que certains améliorent la prise en charge des valeurs de paramètres provenant du pipeline.

Avec cette version il est désormais possible de gérer tous les paramètres communs (Whatif, ErrorAction,…) sans avoir à coder une seule ligne de code, ou encore d'intégrer l'aide dans le code source des fonctions ou des scripts. Elle permet également de créer des proxys de cmdlet et d'organiser votre code à l'aide de module.

Mais commençons par le commencement et avant d'aller plus loin voyons les termes de *paramètre* et d'*argument*.

Pour l'instruction :

```
Get-ChildItem -Path C:\Temp –Recurse
```

-Path est un paramètre et ***C:\temp*** son argument. ***–Recurse*** est un switch qui ne nécessite pas d'argument.

Le runtime PowerShell, au travers des attributs, nous décharge d'une grande partie de la gestion des paramètres d'un script ou d'une fonction, l'objectif étant de proposer un comportement semblable à celui d'un cmdlet tout en facilitant l'écriture du code.

A l'origine l'usage de ces attributs est de contrôler les paramètres d'un cmdlet codé dans un langage dotnet.

Un exemple d'utilisation en C# :

```
[Parameter(Position = 0)]
public string Name
{
  get { return processName; }
  set { processName = value; }
}
private string processName;
```

Ici on précise que le paramètre *Name* se trouve en position zéro. Ainsi, l'appel suivant ne nécessite pas de préciser le nom de l'argument :

```
MonCmdlet –Name "Explorer" –Count 1
 #devient
MonCmdlet "Explorer" –Count 1
```

La version 2 de PowerShell ne propose pas de créer des attributs comme en C#, mais d'utiliser ceux déjà disponibles pour les cmdlets C#. L'équipe de PowerShell a donc ajouté quelques mots-clés spécifiques au contexte de déclaration d'une fonction ou d'un script.

1.1 Principe d'un attribut

Un attribut associe une ou plusieurs informations à un élément du langage. Sous PowerShell seuls les paramètres d'une fonction, d'un script ou d'un Scriptblock sont concernés par les attributs. Leur présence indique au moteur PowerShell de modifier la gestion de ces paramètres, c'est lui qui appliquera les nouveaux comportements précisés par les attributs que vous avez placés sur chacun d'entre eux.

Un attribut peut être considéré comme une métadonnée, c'est-à-dire une information sur une information. Voyons un exemple :

```
Param  (
 [alias("CN")]
 $ComputerName
)
```

On déclare un alias sur le paramètre *$ComputerName.* Il sera donc possible de le référencer, lors d'un appel de cette fonction, par son nom d'origine ou par son nom d'alias.

L'attribut **alias** porte sur le paramètre *$ComputerName*. La déclaration de l'attribut précède celle du nom de paramètre. Sa syntaxe est la suivante :

alias est le nom de l'attribut,

("CN") est un argument renseignant une propriété de cet attribut. Celui-ci peut en déclarer plusieurs, dans ce cas on les séparera par des virgules.

Le tout est déclaré entre crochets **[ ]**.

La syntaxe d'un attribut est donc celle-ci :

```
 [NomAttribut(liste d'arguments)]
Nom_de_paramètre_de_la_fonction
```

Ce qui permet les appels suivants :

```
MaFonctionAvancée -ComputerName "Server1"
 #ou
MaFonctionAvancée –Cn "Server1"
```

On peut également considérer un attribut comme étant un paramétrage, on paramètre du code, ici un paramètre, et pas un traitement. Ce paramétrage est destiné au compilateur du code source PowerShell.

1.2 Les types d'attribut de paramètre

Il existe trois types d'attributs portant sur un paramètre :

1.2.1 Attribut Alias

L'attribut **Alias** spécifie un autre nom pour le paramètre. Le nombre d'alias qui peuvent être affectés à un paramètre est illimité.

Attention les noms d'alias suivants sont réservés : *vb, db, ea, ev, ov,* et *ob*.

1.2.2 Attributs de validation de paramètres

Ces attributs de **validation de paramètres** définissent la façon dont le runtime Windows PowerShell valide les arguments des fonctions avancées.

1.2.3 Attribut Parameter

L'attribut **Parameter** est utilisé pour déclarer un paramètre de la fonction. Cet attribut comporte des arguments nommés utilisés pour définir les caractéristiques du paramètre, par exemple pour savoir s'il est obligatoire ou optionnel.

Il existe également l'attribut **[CmdletBinding()]** qui lui porte sur le comportement d'une fonction ou d'un script, nous l'aborderons par la suite.

1.3 Déclarer un attribut

On déclare un attribut au sein de la clause *Param* :

```
Function FcntAvancée{
  Param (
   [Parameter(Position=0)]
   $NomParam,
   $Count)
  Write-host "`$NomParam=$NomParam `t `$Count=$Count reste=$args"
}
```

La déclaration suivante est autorisée :

```
Function FcntAvancée(
  [Parameter(Position=0)]
  $NomParam,
  $Count){
  Write-host "`$NomParam=$NomParam `t `$Count=$Count reste=$args"
}
```

Cette fonction déclare le paramètre *$NomParam* en précisant qu'il est en première position. Testons notre fonction :

```
FcntAvancée "Deux"
```

```
$NomParam=Deux   $Count= reste=
```

```
FcntAvancée "Deux" 2
```

Le second appel provoque l'erreur suivante :

```
FcntAvancée : Impossible de trouver un paramètre positionnel acceptant l'argument « 2 ».
```

Ceci est du au fait que la présence d'un paramètre positionnel nous contraint à préciser le nom de ceux qui n'ont pas d'indication de position, par exemple *$count* :

```
FcntAvancée "Deux" -count 2
```

```
$NomParam=Deux   $Count=2 reste=
```

Si on ajoute un paramètre « anonyme », récupéré d'habitude dans la variable *$args*, cela provoque la même erreur :

```
FcntAvancée "Deux" -count 2 3
```

```
FcntAvancée : Impossible de trouver un paramètre positionnel acceptant l'argument « 3 ».
```

Modifions l'attribut :

```
Function FcntAvancée{
  Param (
  [Parameter(Mandatory=$True)]
  $NomParam,
  $Count)
  Write-host "`$NomParam=$NomParam `t `$Count=$Count reste=$args"}
```

Cette fois on impose la présence d'un argument pour le paramètre *$NomParam*, testons cette modification :

```
FcntAvancée "Deux" 2
```

```
$NomParam=Deux   $Count=2 reste=
```

```
FcntAvancée "Deux" 2 3
```

```
FcntAvancée : Impossible de trouver un paramètre positionnel acceptant l'argument « 3 ».
```

```
FcntAvancée –count  3
```

```
applet de commande FcntAvancée à la position 1 du pipeline de la commande
Fournissez des valeurs pour les paramètres suivants :
NomParam:
```

On constate que

- le premier appel ne nécessite plus de préciser le nom du second paramètre,
- le second appel provoque toujours une erreur,
- le troisième appel force PowerShell à nous demander une valeur pour le premier paramètre, c'est le comportement attendu.

Enfin pour l'appel suivant PowerShell réorganise la liaison des paramètres :

```
FcntAvancée -count 4 deux
```

```
$NomParam=deux   $Count=4 reste=
```

On voit bien que la présence ou l'absence d'attribut positionnel modifie cette liaison (*Binding Parameters*).

Un autre point concernant la déclaration d'attributs : on peut préciser plusieurs types d'attribut sur chaque paramètre, mais l'attribut **Parameter** doit être unique :

```
Function FcntAvancée{
  Param (
  [Parameter(Mandatory=$True)]
  [Parameter(Position=0)]
  $NomParam,
  $Count)
  Write-host "`$NomParam=$NomParam `t `$Count=$Count"
}
```

```
FcntAvancée deux 4
```
Le paramètre « NomParam » est déclaré plusieurs fois dans le jeu de paramètres « __AllParameterSets ».

Le code précédent provoque une erreur, car l'attribut **Parameter** est précisé deux fois, on doit déclarer plusieurs arguments de la manière suivante :

```
Function FcntAvancée{
  Param (
  [Parameter(Mandatory=$True, Position=0)]
  $NomParam,
  $Count)
  Write-host "`$NomParam=$NomParam `t `$Count=$Count"}
```

2 Liaison de paramètres

Nous avons vu précédemment que la présence d'attribut positionnel modifie la liaison des paramètres. La liaison est l'opération d'affectation d'une valeur à un paramètre, précédée d'autres opérations, telle que la validation des règles portées par les possibles attributs du paramètre.

Il existe un autre attribut qui modifie cette liaison, l'attribut **[CmdletBinding()]** qui doit être impérativement couplé avec une instruction *Param.* Cet attribut précise que notre fonction/script utilisera la même liaison de paramètre qu'un cmdlet, c'est-à-dire que les paramètres inconnus et les arguments positionnels sans paramètre positionnel correspondant entraînent l'échec de la liaison des paramètres en provoquant une exception. Cet attribut autorise d'autres traitements que nous verrons par la suite.

La présence dans une fonction ou un script d'un attribut **Parameter** ou **CmdletBinding** fait que PowerShell déclare une variable automatique nommée *$PSCmdlet* que nous aborderons plus avant dans ce tutoriel.

Sachez qu'une fonction/script déclarant cet attribut n'utilise pas la variable *$args*. Cette variable est toujours présente dans le provider ***variable:***, mais elle référence celle déclarée dans la portée parente. Le test suivant vous permettra de visualiser ce point :

```
Function FcntTest{
  Function FcntAvancée{
   #[CmdletBinding()]
    Param (
   # [Parameter(ValueFromPipeline = $true)]
    $NomParam,
    $Count)
   Write-warning "Dans FcntAvancée "
   if ($myinvocation.MyCommand.CmdletBinding)
    {Write-Host 'CmdletBinding détecté.' -fore Green}
   Write-host "`$NomParam=$NomParam `t `$Count=$Count"
   dir variable:args
   if (test-path Variable:PSCmdlet)
```

```
    {
     Write-Host '$args inexistant localement' -fore Green
     Write-Host "`$PSCmdlet déclaré dans $($pscmdlet.CommandRuntime)"`
     -fore Green
    }
  }

 Function FcntImbriquée {
  [CmdletBinding()]
  Param ([Alias("Test")]
         $ParamTest)

  Write-warning "FcntImbriquée : `$ParamTest=$ParamTest"
  if ($myinvocation.MyCommand.CmdletBinding)
   {Write-Host 'CmdletBinding détecté.' -fore Green}
  dir variable:args
  if (test-path Variable:PSCmdlet)
   {Write-Host 'Liaison modifiée.' -fore Green}
  FcntAvancée deux 10 # 7 "test"
 }

 Write-warning "Dans FcntTest"
 if ($myinvocation.MyCommand.CmdletBinding)
   {Write-Host 'CmdletBinding détecté.' -fore Green}
 dir variable:args
 FcntImbriquée -Test un # 2 34
}
FcntTest un 5 -4 test
```

L'ajout ou la suppression des commentaires, sur tout ou partie des lignes précédentes déclarant un attribut, modifiera le résultat. Je vous laisse tester les différents cas.

Il est possible de tracer le traitement de liaison de la manière suivante :

```
Trace-Command -name ParameterBinding {FcntAvancée "Deux" 2 3} –pshost
```

Notez qu'il existe une variable automatique nommée ***$PSBoundParameters***, pointant sur *$MyInvocation.BoundParameters*, contenant la liste des paramètres liés :

```
Function Test-UnBoundArguments
{
 param([String] $Name,[int]$Nombre=3,[switch] $Stop)
  $OutValue = $null
  if ($MyInvocation.BoundParameters.TryGetValue('Nombre', [ref]$OutValue))
  {
    if ($OutValue -eq $null)
```

```
     {Write-Warning "Le paramètre Nombre prend la valeur `$null."}
    else
     {Write-Warning "Le paramètre Nombre est précisé et prend la valeur $OutValue."}
  }
  else
  {
    Write-Warning "Le paramètre Nombre n'est pas précisé, mais prend la valeur par défaut."
  }
 #if ($PSBoundParameters.ContainsKey('Nombre'))
 # { write-host 'X Bound.' }
"Nombre =$Nombre"
}

Test-UnBoundArguments
Test-UnBoundArguments -name "Test"
Test-UnBoundArguments "Test"
Test-UnBoundArguments "Test" 5
Test-UnBoundArguments "Test" $null
Test-UnBoundArguments "Test" -nombre $null
Test-UnBoundArguments "Test" -nombre
```

Sur le sujet vous pouvez également consulter le schéma suivant, *Cmdlet Processing Lifecycle* :

http://msdn.microsoft.com/en-us/library/ms714429(VS.85).aspx

Si, dans la déclaration d'une fonction, vous ne précisez pas la clause ***Param***, vous devez préciser l'attribut **[CmdletBinding()]** de la manière suivante :

```
Function FcntAvancée(
  [cmdletbinding()]
  [Parameter(Position=0)]
  $NomParam,
  $Count){
  #[cmdletbinding()] -> erreur
  Write-host "`$NomParam=$NomParam `t `$Count=$Count reste=$args"
}
```

Sinon la variable automatique nommée *$PSCmdlet* ne sera pas créée.

2.1 Convention de nommage de Paramètres

Comme pour le nommage des cmdlets, Microsoft recommande de suivre les préconisations portant sur le nommage des noms de paramètres. L'objectif étant de faciliter leur usage auprès des administrateurs, cette convention facilitera la mémorisation et permettra également de deviner rapidement quelles combinaisons utiliser lorsque les administrateurs rencontreront une nouvelle fonction/script avancé.

Le détail de cette convention : *Cmdlet Parameter Name and Functionality Guidelines*

http://msdn.microsoft.com/en-us/library/dd878352(VS.85).aspx

Strongly Encouraged Development Guidelines :

http://msdn.microsoft.com/en-us/library/dd878270(VS.85).aspx

Note : Les noms de paramètre *SelectProperty* et *SelectObject* sont réservés, leur usage générera une erreur :

```
Le nom du paramètre « SelectProperty » est réservé pour une utilisation ultérieure.
Le nom du paramètre « SelectObject» est réservé pour une utilisation ultérieure.
```

3 La gestion du pipeline

L'attribut **Parameter** facilite la liaison d'une valeur d'un paramètre donné avec un objet issu du pipeline. Pour cela on utilise l'argument *ValueFromPipeline* :

```
Function FcntAvancée{
  Param (
  [Parameter(
      Mandatory=$True,
      Position=0,
      ValueFromPipeline = $true)]
  $NomParam,
  $Count)
  Write-host "`$NomParam=$NomParam `t `$Count=$Count"
}
```

On précise ainsi qu'en cas d'utilisation du pipeline l'objet transmis le sera dans la variable *$NomParam*. L'exemple suivant nous montre que la présence de l'argument *ValueFromPipeline* ne met pas pour autant en place les blocs *begin*, *process*, *end* :

```
1,2,3,4|FcntAvancée -count 9
$NomParam=4     $Count=9
```

Il nous faut les préciser, au minimum le bloc *process* :

```
Function FcntAvancée{
  Param (
  [Parameter(
      Mandatory=$True,
      ValueFromPipeline = $true)]
  $NomParam,
  $Count)
  process {
   Write-host "`$NomParam=$NomParam `t `$Count=$Count"
  }
}
1,2,3,4|FcntAvancée -count 9
```

```
$NomParam=1     $Count=9
$NomParam=2     $Count=9
$NomParam=3     $Count=9
$NomParam=4     $Count=9
```

On constate que la liaison fonctionne et la valeur du paramètre *$count* est persistante pour tous les appels du bloc process.

Pour plus de détails sur le sujet, consultez le chapitre 3.2 du tutoriel *L'usage du pipeline sous PowerShell* : http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/Pipelining/

Notez qu'un script .ps1 peut utiliser le pipeline en déclarant les blocs *begin*, *process* et *end*.

3.1 Différences entre la version 1 et la version 2

La gestion des objets issus du pipeline diffère légèrement, prenons l'exemple suivant avec la version 1 de PowerShell :

```
Function FcntAvancéeV1{
  Param (
  $NomParam,
  $Count)
  process {
   Write-host "`$NomParam=$NomParam `t `$Count=$Count"
  }
}
```

L'appel suivant fonctionne :

```
FcntAvancéev1 un 9
$NomParam=un    $Count=9
```

Mais pas celui-ci, car la variable *$NomParam* n'est pas liée :

```
"un"|FcntAvancéev1 -c 9
$NomParam=    $Count=9
```

On doit utiliser **$_** représentant l'objet courant dans le pipeline **:**

```
  process {
   Write-host "`$NomParam=$_ `t `$Count=$Count"
  }
```

Exécutons de nouveau nos deux appels de test, vous remarquerez que cette fois-ci le résultat est inversé, le premier appel ne fonctionne pas, mais le second oui :

```
FcntAvancée un 9
$NomParam=    $Count=9
"un"|FcntAvancée -c 9
$NomParam=un    $Count=9
```

Avec PowerShell version 1 on doit tester, dans une fonction utilisant le pipeline et déclarant les blocs *process* et *end*, si la liaison du paramètre se fait à partir du pipeline ou pas. Sans présence du pipeline c'est le bloc *end* qui est exécuté :

```
Function FcntAvancéeV1{
```

```
  Param (
   $NomParam,
   $Count)

  process {
    if ($_)
     {Write-host "`$NomParam=$_ `t `$Count=$Count"}
  }

  end {
   if ($NomParam)
    {Write-host "`$NomParam=$NomParam `t `$Count=$Count"}
  }
}
```

Le code précédent fonctionne correctement avec les deux exemples d'appels. Il reste un souci qui est que l'on peut utiliser le pipeline tout en précisant le paramètre sur la ligne de commande :

```
"un","Deux"|FcntAvancéeV1 -Nom "Trois" -c 9
```

```
$NomParam=Un      $Count=9
$NomParam=Deux    $Count=9
$NomParam=Trois   $Count=9
```

Dans ce cas, les 2 blocs sont exécutés, pour éviter cela on peut ajouter le test suivant :

```
  process {
    if ($NomParam -and $_)
      {throw "Impossible de coupler l'usage du pipeline avec le paramètre
`$NomParam"}
     …
```

Voyons cet aspect avec la version 2 de PowerShell :

```
Function FcntAvancéeV2{
  Param (
  [Parameter(
      ValueFromPipeline = $true)]
  $NomParam,
  $Count)
  process {
   Write-host "`$NomParam=$NomParam `t `$Count=$Count"
  }
}
```

Testons nos deux appels de test :

```
FcntAvancéeV2 un 9
```

```
$NomParam= un   $Count=9
"un"|FcntAvancéeV2 -c 9
```

```
$NomParam=un    $Count=9
```

Notez qu'avec la version 2, la liaison du nom de paramètre *$NomParam* fonctionne dans les deux cas.

Pour terminer, testons notre cas d'erreur :

```
"Un","Deux"|FcntAvancéeV2 -Nom "Trois" -c 9
```

```
FcntAvancée : L'objet d'entrée ne peut être lié à aucun paramètre de la commande, soit parce que cette commande n'accepte pas l'entrée de pipeline, soit parce que l'entrée et ses propriétés ne correspondent à aucun des paramètres qui acceptent l'entrée de pipeline.
…
```

La version 2 le gère correctement, il déclenche une erreur non bloquante autant de fois qu'il y a d'objets reçus. Le paramètre *$NomParam* étant déjà lié via la ligne de commande, le runtime ne trouve plus de paramètres à lier avec un objet du pipeline et provoque une erreur.

Si on ajoute à notre fonction un bloc *end*, celui-ci sera exécuté, mais si on ne déclare aucun bloc le comportement diffère d'avec la première la version, sans attribut (*FcntAvancéeV1* au début de ce chapitre) :

```
Function FcntAvancéeV2{
  Param (
  [Parameter(ValueFromPipeline = $true)]
  $NomParam,
  $Count)
  Write-host "`$NomParam=$NomParam `t `$Count=$Count"
}

FcntAvancéeV2 un 9
```

```
$NomParam= un   $Count=9
```

```
"un","Deux"|FcntAvancéeV2 -c 9
```

```
$NomParam=Deux    $Count=9
```

Notez que dans le second appel, la valeur *$NomParam* est égale au contenu du deuxième objet émis dans le pipeline, c'est-à-dire égal à la valeur du dernier objet émis.

Testons le cas d'erreur :

```
"Un","Deux"|FcntAvancéeV2 -Nom "Trois" -c 9
```

```
FcntAvancée : L'objet d'entrée ne peut être lié à …

FcntAvancée : L'objet d'entrée ne peut être lié à …
$NomParam=Trois        $Count=9
```

Pour cet exemple, on s'aperçoit que la présence de l'attribut **Parameter**, déclarant *ValueFromPipeline*, déclenche tout de même, à partir des données du pipeline, une itération sur la liaison, alors que le bloc *process* n'existe pas. Si on ne déclare que l'attribut **CmdletBinding** le comportement est identique.

Pour récupérer l'intégralité des données, il nous faut, comme sous PowerShell version 1, utiliser la variable *$Input* et modifier le type $NomParam en un tableau :

```
Function FcntAvancéeV2{
  Param (
```

```
  [Parameter(ValueFromPipeline = $true)]
  [string[]] $NomParam,
  $Count)
 $NomParam=@($input) #Cast d'input en un tableau
 Write-host "`$NomParam=$NomParam `t `$Count=$Count"
}
"un","Deux"|FcntAvancéeV2 -c 9
```

```
$NomParam=un Deux       $Count=9
```

L'exemple suivant met en évidence un bug référencé, l'usage d'une fonction au sein d'une sous expression **$(…)** provoque des itérations erronées :

```
function Get-Something {
 Write-Host 'Got something'
 'Something'
}
function test {
  [CmdletBinding()] # <- bug
  param(
  [Parameter(Position=0)]
  [string]
  $Value = $(Get-Something),
  [Parameter(ValueFromPipeline=$true)]
  [PSObject]
  $PipeObject
  )
  begin{
    Write-Host "Changing the value from '$Value' to 'Something else'." -
ForegroundColor Green
    $Value = 'Something else'
  }
  process{
    $Value
  }
}
```

Le problème est qu'en interne la liaison du paramètre se fait 5 fois, la gestion des attributs de validation de PowerShell n'est pas impactée, eux sont bien exécutés une seule fois.

Voir le détail du bug sur MSConnect :

https://connect.microsoft.com/PowerShell/feedback/ViewFeedback.aspx?FeedbackID=509985

4 Les arguments de l'attribut Parameter

Ce chapitre reprend et complète les informations du fichier *about_Functions_Advanced_Parameters.txt*

Tous ces arguments sont optionnels, ceux étant de type booléen ont par défaut la valeur *$false*.

4.1 Mandatory

L'argument **Mandatory**, de type booléen, indique que le paramètre est obligatoire lorsque la fonction est exécutée. Si cet argument n'est pas spécifié, le paramètre est alors un paramètre optionnel.

Note : Passer en argument la valeur *$null* à un paramètre obligatoire déclenchera une exception.

Attention, si un paramètre portant cet argument déclare une valeur par défaut, son caractère obligatoire persiste. C'est-à-dire que la présence de ce paramètre sur la ligne de commande sera toujours obligatoire.

4.2 Position

L'argument **Position**, de type entier, spécifie la position du paramètre. Si cet argument n'est pas spécifié, le nom de paramètre ou son alias doit être spécifié explicitement quand la valeur du paramètre est définie.

De même, si aucun des paramètres d'une fonction n'a de position, le runtime Windows PowerShell affecte des positions à chaque paramètre selon l'ordre dans lequel ils sont reçus.

Note : Il est possible de déclarer plusieurs paramètres déclarants une position identique, en cas d'ambiguïté PowerShell déclenchera une exception :

```
Impossible de lier les paramètres positionnels, car aucun nom n'a été fourni.
```

Les nombres indiquant la position peuvent ne pas être ordonnés ni se suivre :

```
Function TestProperty {
  [CmdletBinding()]
  Param (
    [Parameter(Position=7)]
   $Count,
    [Parameter(Position=1)]
   $Name,
    [Parameter(Position=5)]
   $test)

   "Name = $Name"
   "Test = $test"
   "Count = $Count"
 }
TestProperty un deux trois
```

```
Name = un
```

```
Test = deux
Count = trois
```

4.3 HelpMessage

L'argument **HelpMessage**, de type String, spécifie un message contenant une description courte du paramètre. Pour la déclaration suivante :

```
[Parameter(Position=0,Mandatory=$true,HelpMessage="Objet à analyser.")]
$NomParam="Defaut"
```

En cas d'absence du paramètre, PowerShell proposera de le saisir et affichera le texte suivant :

```
applet de commande FcntAvancée à la position 1 du pipeline de la commande
Fournissez des valeurs pour les paramètres suivants :
(Tapez !? pour obtenir de l'aide.)
NomParam: !?
Objet à analyser.
NomParam:
```

La saisie des caractères **!?**, suivis d'un retour chariot, affichera le texte d'aide *Objet à analyser*.

Il existe deux autres arguments reconnus pour l'attribut Parameter permettant de définir le message d'aide à partir d'un fichier de ressources :

```
…
#Les valeurs utilisées ici existent bien
[Parameter(Mandatory=$true,HelpMessageBaseName="ParameterBinderStrings",
HelpMessageResourceId="PositionalParameterNotFound")] $Test
```

Malheureusement, la gestion interne de PowerShell n'autorise pas d'ajouter un fichier de ressources additionnelles afin de gérer ces arguments dans une fonction avancée. Un petit souci en cas de localisation de cet argument.

4.4 ValueFromPipeline

L'argument **ValueFromPipeline**, de type booléen, spécifie que la valeur du paramètre peut provenir de l'objet émis dans le pipeline. Spécifiez cet argument si la fonction ou le script accède à l'objet complet et pas seulement à une propriété de l'objet.

Par convention, le nom standard d'un paramètre lié au pipeline est ***InputObject***.

Note : Il est possible de déclarer plusieurs paramètres avec cet argument, dans ce cas chacun de ces paramètres recevra la même référence sur l'objet.

Voir aussi : *Understanding ByValue Pipeline Bound Parameters*

http://keithhill.spaces.live.com/Blog/cns!5A8D2641E0963A97!6158.entry

4.5 ValueFromPipelineByPropertyName

L'argument **valueFromPipelineByPropertyName**, de type booléen, spécifie que la valeur du paramètre peut provenir d'une propriété de l'objet émis dans le pipeline. Spécifiez cet attribut si les conditions suivantes sont remplies :

- Le paramètre accède à une propriété de l'objet émis dans le pipeline.
- La propriété de l'objet émis porte le même nom que le paramètre, ou la propriété a le même nom qu'un des alias du paramètre.

L'exemple suivant teste les deux cas :

```
Function TestProperty {
  [CmdletBinding()]
  Param (
   [Parameter(ValueFromPipelineByPropertyName = $true,
              HelpMessage = "Objet à analyser.")]
   [Alias('Nom')]
   [System.String] $Name)
  begin {
     $StringFormat= "[Nom de paramètre] {0}"
  } #Begin
  process {
    $StringFormat -F $Name
  }#process
}  #TestProperty
```

L'accès par nom de propriété, on crée un objet possédant une propriété nommée *Name* :

```
$hash=@{};
$hash.Name="Test avec Name"
$object = new-object PSObject -property $hash
$object|TestProperty
```

```
[Nom de paramètre] Test avec Name
```

L'accès par alias de nom de propriété, on crée un objet possédant une propriété nommée *Nom* :

```
$hash=@{};
$hash.Nom="Test avec Nom"
$object = new-object PSObject -property $hash ; $object|TestProperty
```

```
[Nom de paramètre] Test avec Nom
```

Si on crée un objet ne possédant pas de propriété nommée *Name* ou *Nom* :

```
$hash=@{};
$hash.Non="Test avec Non.Aucune correspondance."
$object = new-object PSObject -property $hash ; $object|TestProperty
```

Alors, la tentative de liaison à partir du pipeline générera l'exception suivante :

```
TestProperty : L'objet d'entrée ne peut être lié à aucun paramètre de la commande, soit parce que cette commande n'accepte pas l'entrée de pipeline, soit parce que l'entrée et ses propriétés ne correspondent à aucun des paramètres qui acceptent l'entrée de pipeline.
```

Toutes les classes existantes possédant une propriété *Name* pourront être utilisées :

```
Dir c:\temp|TestProperty
```

```
[Nom de paramètre] Adobe
```

```
[Nom de paramètre] Google Toolbar
[Nom de paramètre] VBE
…
```

Il est possible de récupérer plusieurs propriétés :

```
Function TestProperty {
  [CmdletBinding()]
  Param (
   [Parameter(ValueFromPipelineByPropertyName = $true,
            HelpMessage = "Objet à analyser.")]
   [Alias('Nom')]
   [System.String] $Name,

   [Parameter(ValueFromPipelineByPropertyName = $true)]
   [Alias('Length')]
   $Count)

  begin {
    $StringFormat= "[Nom de paramètre] {0} [Count] {1}"
  } #Begin
  process {
    $StringFormat -F $Name,$Count
  }#process
} #TestProperty

$hash=@{};
$hash.Name="Test avec Name"
$hash.Count=5
$object = new-object PSObject -property $hash ; $object|TestProperty
```

```
[Nom de paramètre] Test avec Name [Count] 5
```

```
Dir|Where {!$_.PSIsContainer}|TestProperty
```

```
[Nom de paramètre] A propos de Add-Lib.url [Count] 140
[Nom de paramètre] Revisions.txt [Count] 4935
…
```

Notez qu'il n'est pas nécessaire de coupler cet argument avec ***ValueFromPipeline***. Un seul suffit, mais les deux peuvent être couplés sur le même paramètre. Si vous le faites et que l'objet ne possède pas de propriété de même nom, le paramètre sera alors lié non plus à la propriété de l'objet émis dans le pipeline, mais à l'objet, si toutefois le type du paramètre le permet.

Quand un paramètre est lié par *ByValue* et par *ByPropertyName*, PowerShell essaie de le lier dans l'ordre suivant :

- ✓ liaison par *ByValue* sans conversion de types,
- ✓ liaison par *ByPropertyName* sans conversion de types,

- ✓ liaison par *ByValue* avec conversion de types,
- ✓ liaison par *ByPropertyName* avec conversion de types,

Une transformation implicite, un cast, peut donc avoir lieu lors de la liaison. Vous pouvez tester ces comportements en modifiant le type du paramètre *$Name* :

```
Function TestProperty {
  Param (
    [Parameter(
          ValueFromPipelineByPropertyName = $true,
          ValueFromPipeline = $true)]
   #[String]$Name    # cast implicite de l'objet String
   [Int]$Name        # cast implicite de l'objet Int
   #$Name            # tout objet est accepté
   )

  Process { "[Nom de paramètre] $Name" ; $Name.GetType();"`r`n"}
}
$hash=@{}; $hash.Name="Test"
$object = new-object PSObject -property $hash
$hash=@{}; $hash.Name=3
$object2 = new-object PSObject -property $hash
$cp = new-object Microsoft.CSharp.CSharpCodeProvider
$object,$object2,$cp,10,3.5,$null,"" |TestProperty
```

On passe un objet personnalisé possédant une propriété *Name*, un objet dotnet et un integer :

```
$object,$object2,$cp,10,3.5,$null,""|TestProperty
```

Dans le cas où la conversion ne peut se faire, PowerShell déclenche une erreur non bloquante. C'est le cas pour le type [int] :

```
TestProperty : Impossible de traiter la transformation d'argument sur le paramètre « Name ». Impossible de
convertir la valeur « Test » en type « System.Int32 ». Erreur : « Le format de la chaîne d'entrée est incorrect. »

[Nom de paramètre] 3
IsPublic IsSerial Name                     BaseType
-------- -------- ----                     --------
True     True     Int32                    System.ValueType

TestProperty : L'objet d'entrée ne peut être lié à aucun paramètre de la commande, soit parce que cette
commande n'accepte pas l'entrée de pipeline, soit parce que l'entrée et ses propriétés ne correspondent à aucun
des paramètres qui acceptent l'entrée de pipeline.

[Nom de paramètre] 10
True     True     Int32                    System.ValueType

[Nom de paramètre] 4
True     True     Int32                    System.ValueType
```

```
[Nom de paramètre]
Vous ne pouvez pas appeler de méthode sur une expression ayant la valeur Null.

[Nom de paramètre] 0
True    True    Int32                   System.ValueType
```

Vous noterez également que pour le type [int], une chaîne vide se voit attribuer une valeur par défaut, et que la valeur [Double] est arrondie.

Étant donné que l'on utilise le plus souvent une seule information d'un objet, l'usage couplé de ces deux arguments permet d'augmenter les possibilités de réussite lors de la liaison, sous réserve que ce couplage ait un sens.

Voir aussi : *Understanding ByPropertyName Pipeline Bound Parameters*

http://keithhill.spaces.live.com/Blog/cns!5A8D2641E0963A97!6130.entry

4.6 ValueFromRemainingArguments

L'argument **ValueFromRemainingArguments**, de type booléen, spécifie que le paramètre accepte tous les arguments restants qui ne sont pas liés aux paramètres de la fonction :

```
Function FcntAvancée{
  #[CmdletBinding()]
  Param (
   [Parameter(Position=0)]
   $NomParam,
   $Count,
   [parameter(ValueFromRemainingArguments=$true)]
   [String[]] $ArgsRestant)
   Write-host "`$NomParam=$NomParam `t `$Count=$Count Reste=$ArgsRestant"
}
FcntAvancée "Deux" 1 2 3 4
$NomParam=Deux   $Count= Reste=1 2 3 4
```

Note : A partir du moment où on utilise les fonctions avancées, on doit préciser nos intentions.

4.7 ParameterSetName

L'argument **ParameterSetName**, de type String, spécifie le jeu de paramètres auquel un paramètre appartient. Cet argument implémente une sorte de regroupement fonctionnel de paramètres. Il permet différentes utilisations d'une fonction/script avancés selon l'appartenance d'un paramètre à tel ou tel jeu de paramètres.

Un paramètre ne déclarant aucune appartenance à un jeu de paramètres (*ParameterSet)* appartient à tous les groupes existants dans la fonction ou script.

Un paramètre déclarant une appartenance à un jeu de paramètre peut appartenir à plusieurs jeux, mais chaque jeu doit avoir au moins un paramètre unique. Chaque jeu contient au moins un paramètre.

Prenons par exemple le cmdlet compilé **Get-EventLog**, en utilisant Reflector.exe on voit qu'il utilise deux jeux de paramètres, *List* et *LogName* :

Essayons la combinaison de paramètre suivante :

```
Get-Eventlog -list -logname "System"
```

```
Get-EventLog : Le jeu de paramètres ne peut pas être résolu à l'aide des paramètres nommés spécifiés.
```

L'usage de paramètres utilisant plusieurs jeux de paramètres provoque une exception. Fonctionnellement on peut utiliser le cmdlet **Get-EventLog** soit en mode liste, on obtient tous les journaux de logs, soit en mode nom de journal et dans ce cas on obtient les entrées du journal indiqué, mais on ne peut pas utiliser les deux traitements en même temps.

Implémentons une fonction offrant le même comportement que cmdlet **Get-EventLog** (on implémente uniquement les paramètres et leurs attributs) :

```
Function GetEventLog{
 [CmdletBinding(DefaultParameterSetName="LogName")]
  Param (
   [Parameter(ParameterSetName="List")]
   [Switch] $AsString,
   [Parameter(ParameterSetName="List")]
   [Switch] $List,
   [Parameter(Position=0, Mandatory=$true, ParameterSetName="LogName")]
   [string] $LogName,
   [Parameter(ParameterSetName="LogName")]
   [ValidateRange(0, 0x7fffffff)]
   [int] $Newest)

    Write-Host "Traitement..."
}
Geteventlog -list -logname "System"
```

On obtient le même message d'erreur.

Autre exemple avec le cmdlet **Get-WMIObject** :

Disassembler

```
[Cmdlet("Get", "WmiObject", DefaultParameterSetName="query")]
public class GetWmiObjectCommand : PSCmdlet
{
  // Properties
  [Parameter(Position=0, Mandatory=true, ParameterSetName="query")]
  public string Class { get; set; }
  [Parameter, ValidateNotNull]
  public string[] ComputerName { get; set; }
  [Parameter, Credential]
  public PSCredential Credential { get; set; }
  [Parameter(ParameterSetName="query")]
  public string Filter { get; set; }
  [Parameter(ParameterSetName="list")]
  public SwitchParameter List { get; set; }
  [Parameter]
  public string Namespace { get; set; }
  [Parameter(Position=1, ParameterSetName="query")]
  public string[] Property { get; set; }
  [Parameter(Mandatory=true, ParameterSetName="WQLQuery")]
  public string Query { get; set; }
}
```

Ce cmdlet déclare trois jeux de paramètres et certains paramètres n'appartiennent explicitement à aucun jeu, implicitement ils appartiennent à chaque jeu de paramètres déclaré.

4.7.1 Validité des jeux de paramètres

Dans l'illustration suivante, la colonne de gauche montre trois ensembles valides de paramètres :

Valid Parameter Sets	Invalid Parameter Sets
A D B D E C D	A B B C A C

© Microsoft MSDN

Le paramètre **A** est unique et appartient au premier ensemble de paramètres, le paramètre **B** est unique et appartient au deuxième ensemble de paramètres, et le paramètre **C** est unique et appartient au troisième ensemble de paramètres. Le paramètre **D** appartient à tous les ensembles de paramètres. Le paramètre **E** appartient uniquement au deuxième ensemble de paramètres.

Cependant, dans la colonne de droite, aucun des ensembles de paramètres ne déclare de paramètre unique, ils sont donc invalides. L'unicité d'au moins un paramètre dans un jeu permet au runtime de déterminer quelle fonctionnalité on utilise.

Testons ces deux cas, d'abord les ensembles invalides :

```
Function InvalideParameterSet{
  Param (
   [Parameter(ParameterSetName="Fonctionnalite1")]
   [Parameter(ParameterSetName="Fonctionnalite3")]
   [Switch] $A,
   [Parameter(ParameterSetName="Fonctionnalite1")]
```

```
   [Parameter(ParameterSetName="Fonctionnalite2")]
   [Switch] $B,
   [Parameter(ParameterSetName="Fonctionnalite2")]
   [Parameter(ParameterSetName="Fonctionnalite3")]
   [Switch] $C)

   Write-Host "Traitement..."
}
```

Les appels suivants ne fonctionnent pas, car PowerShell ne sait pas de quel jeu de paramètres il s'agit. Par exemple pour **–A** est-ce qu'il s'agit de la *Fonctionnalite1* ou de la *Fonctionnalite3* ?

```
InvalideParameterSet –A; InvalideParameterSet –B; InvalideParameterSet –C
```

Ceux-ci fonctionnent car on sait de quel jeu il s'agit :

```
InvalideParameterSet –A -B; InvalideParameterSet –B -C;
InvalideParameterSet -A –C
```

L'appel suivant ne fonctionne pas :

```
InvalideParameterSet
```

```
InvalideParameterSet : Le jeu de paramètres ne peut pas être résolu à l'aide des paramètres nommés spécifiés.
```

À partir du moment où on déclare au moins deux jeux de paramètres, on doit en spécifier au moins un en précisant un paramètre sur la ligne de commande. Si on implémente plusieurs fonctionnalités, le runtime doit savoir laquelle utiliser, il ne peut pas le deviner !

Vous remarquerez que, bien que ces jeux de paramètres soient considérés comme invalides, il reste possible de les utiliser, il y a "juste" une incohérence dans la mise en œuvre.

Testons maintenant les ensembles valides :

```
Function ValideParameterSet{
  Param (
   [Parameter(ParameterSetName="PSet1")]
   [Switch] $A,
   [Parameter(ParameterSetName="PSet2")]
   [Switch] $B,
   [Parameter(ParameterSetName="PSet3")]
   [Switch] $C,
   [Switch] $D,
   [Parameter(ParameterSetName="PSet2")]
   [Switch] $E)

   Write-Host "Traitement..."
}
```

Ici la présence du paramètre unique dans chaque jeu de paramètre permet au runtime de déterminer quelle fonctionnalité utiliser :

```
ValideParameterSet –A ; ValideParameterSet –B; ValideParameterSet –C
```

Les appels suivants sont corrects :

```
ValideParameterSet -A –D; ValideParameterSet -B -E
```

Mais pas ceux-ci :

```
ValideParameterSet -A –C
```

```
InvalideParameterSet : Le jeu de paramètres ne peut pas être résolu à l'aide des paramètres nommés spécifiés.
```

```
ValideParameterSet –D
```

```
InvalideParameterSet : Le jeu de paramètres ne peut pas être résolu à l'aide des paramètres nommés spécifiés.
```

On utilise des paramètres uniques appartenant à de jeux différents ou un paramètre appartenant à tous les jeux. Le paramètre ***–D*** ne permet pas à lui seul de déterminer la fonctionnalité que l'on souhaite utiliser.

Enfin pour l'appel suivant on obtient la même erreur :

```
ValideParameterSet
```

Pour l'éviter, on doit préciser, à l'aide de l'attribut **[CmdletBinding()],** quel jeu de paramètre on utilisera par défaut :

```
Function ValideParameterSet{
  [CmdletBinding(DefaultParameterSetName="PSet2")]
  Param (
   [Parameter(ParameterSetName="PSet1")]
   [Switch] $A,
…
```

Ceci fait, on peut exécuter notre fonction/script sans préciser de paramètre et autoriser un comportement par défaut. On peut tester le nom du jeu de paramètre en cours à l'aide de la variable automatique *$PSCmdlet* :

```
switch ($PsCmdlet.ParameterSetName)
 {
   "PSet1" { Write-Host "Fonctionnalité 1 (PSet1)"; break}
   "PSet2" { Write-Host "Fonctionnalité 2 (PSet2)"; break}
   "PSet3" { Write-Host "Fonctionnalité 3 (PSet3)"; break}
 }
```

Une fois déclaré le jeu de paramètre par défaut, l'appel suivant fonctionne :

```
ValideParameterSet –D
```

```
Fonctionnalité 2 (PSet2)
```

La présence de l'argument *DefaultParameterSetName* lève toutes ambiguïtés.

4.7.2 L'attribut OutputType

Dans la nouvelle version de son ouvrage sur PowerShell, Bruce Payette mentionne l'existence d'un attribut non documenté nommé **OutputType**.Son rôle consiste à préciser le type de la valeur de retour associée au jeu de paramètre actif :

```
Function ValideParameterSet{
 [CmdletBinding(DefaultParameterSetName="PSet2")]
```

```
[OutputType("PSet1", [int])]
[OutputType("PSet2", [Byte[]])]
[OutputType("PSet3", [String])]
Param (
 [Parameter(ParameterSetName="PSet1")]
 [Switch] $A,
  …
```

Mais dans la version 2, son usage ne modifie pas la valeur de retour, en l'état il permet de documenter le code.

Voir aussi :

PowerShell V2: ParameterSets

http://blogs.msdn.com/powershell/archive/2008/12/23/powershell-v2-parametersets.aspx

Adding Parameter Sets to a Cmdlet

http://msdn.microsoft.com/en-us/library/ms714647(VS.85).aspx

Dans le projet PSCX, le code source du cmdlet **Get-DomainController** contient un exemple de gestion de switchs exclusif basé sur l'usage de cet argument.

5 Les attributs de validation

Ce chapitre reprend et complète des informations du fichier *about_Functions_Advanced_Parameters.txt*

Ces attributs définissent la façon dont le runtime Windows PowerShell valide les arguments des fonctions/scripts avancées.

Ils sont identiques à ceux présentés dans le tutoriel *Les variables contraintes sous PowerShell* : http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/VariablesContraintes/

Ces attributs peuvent exister indépendamment de l'attribut **Parameter :**

```
Function TestValidation{
  Param (
    [AllowNull()]
    [ValidateRange(5,20)]
   [Int] $Number)
  "$Number Valide"
}
```

Dans ce cas, si vous souhaitez utiliser la variable *$PSCmdlet*, vous devrez préciser l'attribut **[CmdletBinding()]**.

Si vous placez plusieurs attributs de validation sur un paramètre, tous les tests de validation doivent réussir. En cas d'échec PowerShell déclenche une exception du type :

```
System.Management.Automation.ParameterBindingValidationException
```

Notez également que la règle de validation est persistante, le paramètre devient donc une variable contrainte. Si on interdit d'affecter la valeur *$null* à un paramètre, cette règle reste valide dans le code de la fonction ou du script pour la variable associée à ce paramètre.

Le paramètre doit être lié pour que ses règles de validation se déclenchent. Si vous liez la valeur d'un paramètre, ou plusieurs, *via* le pipeline, ses règles associées se déclencheront à chaque opération de liaison.

La validation n'est pas déclenchée pour les autres paramètres dont les valeurs ne changent pas.

Dans l'exemple suivant l'absence d'arguments, lors de l'appel de la fonction, génère une chaîne de caractères vide et un tableau de valeur *$null* :

```
Function TestValidation {
  Param
   (
      #[parameter(Mandatory=$true)]
      [ValidateNotNullOrEmpty()]
      [String] $userName,
      [ValidateNotNullOrEmpty()]
      [String[]] $Tab
    )
  "Paramètres valides"
  "  `$username -eq [string]::Empty : $($username -eq [string]::Empty)"
  "  `$username -eq `$null : $($username -eq $null)"
  #$username=[string]::Empty #exception
  "  `$Tab -eq [string]::Empty : $($Tab -eq [string]::Empty)"
  "  `$Tab -eq `$null : $($Tab -eq $null)"
}
TestValidation
```

```
Paramètres valides
  $username -eq [string]::Empty : True
  $username -eq $null           : False
  $Tab -eq [string]::Empty      : False
  $Tab -eq $null                : True
```

C'est un peu déroutant, mais on ne peut pas valider une valeur optionnelle qui n'est pas spécifiée lors de l'appel de la fonction/script.

Pour forcer la validation des paramètres on peut placer les affectations suivantes dans les premières lignes du code de la fonction/script :

```
  $username=$username
  $tab=$tab
```

La différence étant que PowerShell génère cette fois-ci non pas une exception, mais une erreur non bloquante de type *ValidationMetadataException* :

```
La variable ne peut pas être validée, car la valeur n'est pas une valeur valide pour la variable userName.
…
La variable ne peut pas être validée, car la valeur n'est pas une valeur valide pour la variable Tab.
```

```
…
```
On peut donc dire que PowerShell valide dans un premier temps un paramètre, puis la variable associée.

Les appels suivants sont correctement gérés :

```
TestValidation $null
TestValidation –UserName $null
```
```
TestValidation : Impossible de valider l'argument sur le paramètre « userName ». L'argument est null ou vide. Indiquez un argument qui n'est pas null ou vide et réessayez.
```
Mais si vous modifiez, pour le paramètre *$UserName*, l'attribut ***ValidateNotNullOrEmpty*** en ***ValidateNotNull***, alors PowerShell effectue un cast avant d'exécuter la validation, ce qui fait que la validation de ce paramètre réussi.

Ceci est dû aux règles de transtypage :

```
($null -as [string]) -eq [String]::Empty
```
```
True
```
```
$S=$null -as [string]
$S -eq [String]::Empty
```
```
True
```
```
($null -as [Double]) -eq 0
```
```
True
```
```
$D=$null -as [Double]
$D -eq 0
```
```
True
```
L'opérateur **–as** affecte implicitement la valeur par défaut du type utilisé lors de l'opération.

Ces attributs de validation constituent une sorte de contrat sous forme de préconditions.

5.1 AllowNull

L'attribut **AllowNull** autorise l'argument d'un paramètre obligatoire à être affectée avec la valeur *$Null*. Cet attribut ne nécessite pas d'argument.

```
[AllowNul()]
```

5.2 ValidateNotNull

L'attribut **ValidateNotNull** spécifie que l'argument du paramètre ne peut pas être défini avec *$Null*. Le runtime Windows PowerShell génère une erreur si la valeur du paramètre est *$Null*.

Cet attribut ne nécessite pas d'argument.

Si le paramètre est une collection, celle-ci ne doit pas contenir d'élément de valeur *$null* :

```
Function TestValidation {
  Param
   (
    [ValidateNotNull()]
    [String[]] $Objets
    )
```

```
  "Paramètre valide"
  $Objets
}
TestValidation @(1,(Get-Date),$null)
```

Les collections utilisables avec cet attribut doivent implémenter au moins une des interfaces suivantes : *IList, ICollection, IEnumerable, IEnumerator*. Notez que la valeur *$null* est transformée en chaîne vide.

Pour retrouver les interfaces implémentées :

```
$PSVersiontable.GetType().GetInterfaces()
```

5.3 AllowEmptyString

L'attribut **AllowEmptyString** autorise une chaîne vide comme argument d'un paramètre obligatoire. Cet attribut ne nécessite pas d'argument.

5.4 ValidateNotNullOrEmpty

L'attribut **ValidateNotNullOrEmpty** spécifie que l'argument du paramètre ne peut pas être défini avec *$Null*, ni être vide. Le runtime Windows PowerShell génère une erreur si le paramètre est spécifié et que sa valeur est soit *$Null*, soit une chaîne vide ou un tableau vide.

Cet attribut ne nécessite pas d'argument.

Si le paramètre est une collection, celle-ci ne doit pas contenir d'élément de valeur *$null* ou vide :

```
Function TestValidation {
  Param
   (
    [ValidateNotNullOrEmpty()]
    [Object[]] $Objets
    )
  "Paramètre valide"
  $Objets
}
TestValidation @(1,(Get-Date),$null)
```

Les collections doivent implémenter au moins une des interfaces suivantes : *IList, ICollection, IEnumerable, IEnumerator*.

A la différence de l'attribut **ValidateNotNull**, ici le même code génère une exception.

5.5 AllowEmptyCollection

L'attribut **AllowEmptyCollection** autorise une collection vide comme argument d'un paramètre obligatoire. Cet attribut ne nécessite pas d'argument.

5.6 ValidateCount

L'attribut **ValidateCount** spécifie le nombre minimal et le nombre maximal d'éléments, tous deux de type integer, que le paramètre, de type collection, peut accepter. Le runtime Windows PowerShell génère une erreur si le nombre d'éléments se trouve à l'extérieur de cette plage :

```
 [ValidateCount(1,26)] # de A à Z
[Char[]] $HardDrives
```

5.7 ValidateLength

L'attribut **ValidateLength** spécifie la longueur minimale et la longueur maximale, tous deux de type integer, de la valeur du paramètre (celui-ci est en interne transformé en **[string]**). Le runtime Windows PowerShell génère une erreur si la longueur de la valeur du paramètre se trouve à l'extérieur de cette plage. La variable peut être de type string ou tableau de string :

```
Function TestValidation {
  Param
   (
    [ValidateLength(3,5)]
    #[String[]] $T
    [String] $T

    )
  "Paramètre valide"
  $T
}
TestValidation "100"
#TestValidation @("100","1457",$null)
```

On peut par exemple passer en paramètre un tableau de char à condition de modifier la variable ***$OFS*** :

```
Function TestValidation {
  Param(
     [ValidateLength(3,5)]
    [String] $T
   )
  "Paramètres valides"
  $T
}
TestValidation "Test"
[Char[]] $Chars=@(65,66,67,68,69)
  #---Erreur dû à $OFS
"$Chars".length
  #Cast explicite en [String]
```

```
TestValidation  $Chars

$ofs=""
"$Chars".length
TestValidation  $Chars
 #---Erreur de validation de la longueur
[Char[]] $Chars=@(65,66,67,68,69,70)
TestValidation  $Chars
```

5.8 ValidatePattern

L'attribut **ValidatePattern** spécifie une expression régulière validant le contenu du paramètre de type string. Le runtime Windows PowerShell génère une erreur si la valeur du paramètre ne correspond pas au modèle de l'expression régulière :

```
 [ValidatePattern('^([A-Za-z]{3})$')]   # Mot de trois lettres
[String] $TriGram="ABC"
```

5.9 ValidateRange

L'attribut **ValidateRange** spécifie les valeurs minimales et maximales, tous deux de type object, de la valeur du paramètre, lui aussi de type **[Object].** Le runtime Windows PowerShell génère une erreur si la valeur du paramètre se trouve à l'extérieur de cette plage :

```
 [ValidateRange("A","Z")]
[Char] $DriveName
```

Attention cet attribut est inadapté (buggé ?) pour certains types, par exemple il est impossible de valider une étendue de date. Pour contourner ce problème, on utilisera l'attribut **ValidateScript**.

5.10 ValidateSet

L'attribut **ValidateSet** spécifie un jeu de valeurs valides pour la valeur du paramètre, qui peut être un tableau. Le runtime Windows PowerShell génère une erreur si la valeur du paramètre ne correspond pas à une des valeurs de ce jeu :

```
 [ValidateSet("HKCR","HKCU","HKLM","HKCC")]
[String] $HiveShortCut
```

Ici on parcourt et valide chaque élément du tableau de chaîne de caractères passée en paramètre :

```
  [ValidateSet("HKCR","HKCU","HKLM","HKCC")]
#[String[]] $HiveShortCut
```

Le type des valeurs de l'ensemble peut être de tout type scalaire, mais sous forme de constante :

```
Function TestValidation {
  Param
   (
    [ValidateSet(5,7,9,12)]
    #[Int] $Nb
```

```
    [Int[]] $Nb
    )
  "Paramètre valide"
   $Nb
}

TestValidation @(5,1)
```

5.11 ValidateScript

L'attribut **ValidateScript** spécifie un Scriptblock validant le contenu du paramètre. Le runtime Windows PowerShell génère une erreur si le résultat du Scriptblock est faux ou si le Scriptblock lève une exception.

Dans tous les cas la valeur à tester au sein du Scriptblock provient du pipeline. Sachez que la variable associée au paramètre n'est pas encore déclarée lorsque ***Validatescript*** est appelé, on ne reçoit que la valeur du paramètre.

Dans l'exemple suivant, la valeur du paramètre *Count* doit être inférieure à 4 :

```
Function Test{
    Param (
      [ValidateScript({$_ -lt 4})]
      [int] $Count
    )
 $count
}
Test 5
```

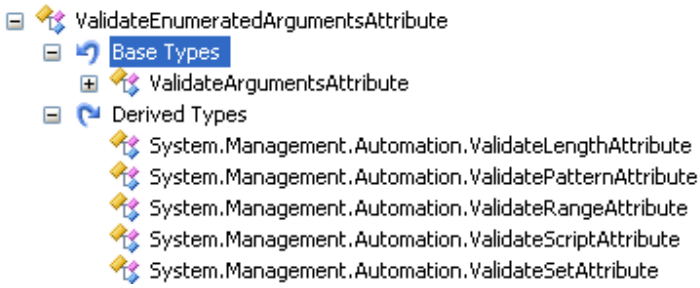
Il est possible de faire appel à script ou à une fonction existante :

```
function FnctValidation()
{$_ -lt 4}

Function Test{
    Param (
       [ValidateScript({FnctValidation})]
       #[ValidateScript({&C:\temp\TestValidation.ps1})]
      [int] $Count)
  $count
}
Test 5
```

Rappel :

Les attributs dérivés suivants partagent le même comportement sur les variables énumérables, c'est-à-dire de type collection :



Enfin sachez que les erreurs déclenchées par un attribut de validation ne sont pas bloquantes :

```
Function Test{
    Param (
         [Parameter(ValueFromPipeline = $true)]
       [ValidateScript({$_ -lt 4})]
      [int] $Count)
  process  { $count }
}
1,5,3,7,9,2|Test
```

```
1
Test : Impossible de valider l'argument sur le paramètre « Count ». Le script de validation « $_ -lt 4 » pour l'argument avec la valeur « 5 » n'a pas retourné une valeur True. Déterminez pourquoi la validation du script a échoué et réessayez.
…
3
Test : Impossible de valider l'argument sur le paramètre « Count ». Le script de validation « $_ -lt 4 » pour l'argument avec la valeur « 7 » …
 Test : Impossible de valider l'argument sur le paramètre « Count ». Le script de validation « $_ -lt 4 » pour l'argument avec la valeur « 9 » …
2
```

N’ayant pas codé de gestion des erreurs dans le Scriptblock, PowerShell utilise un message prédéfini.

5.11.1 Création de règles de validation

L’attribut **ValidateScript** peut être utilisé pour créer des règles personnelles de validation d’arguments de paramètre ou pour tracer ses modifications.

Prenons le cas où l’on veuille imposer la règle suivante : un chemin ne doit pas contenir de caractères du globbing, c'est-à-dire *, ? et [] :

```
Function Test-AcceptsWildcards{
 if ($MyInvocation.CommandOrigin -eq "Internal")
  { Write-Debug "Exécution via un attribut" }
 else
  { Write-Debug "Exécution via un runspace" }
 If (
([Management.Automation.WildcardPattern]::ContainsWildcardCharacters($_)))
  {
```

```
    $VMEx="System.Management.Automation.ValidationMetadataException"
    $EGlobbing="Le globbing (?,*,[]) n'est pas supporté ({0})."
   Throw (new-object $VMEx ($EGlobbing -F $_))
  }
 #La valeur est valide
 $true }
```

Le premier test sur la variable *$MyInvocation.CommandOrigin* détermine si le code est exécuté par le runtime *via* un attribut ou *via* une affectation dans un runspace.

```
function Test( [ValidateScript( {Test-AcceptsWildcards} )]
               [Parameter(ValueFromPipeline = $true)] $Path)
{ $Path }
"C:\*.exe"|Test
```

Ainsi, on obtient l'erreur suivante :

```
Test : Impossible de valider l'argument sur le paramètre « Path ». "Le globbing (?,*,[]) n'est pas supporté (C:\*.exe).
```

5.11.2 Usage de fonction de validation dans un module

À des fins de réutilisation, il est possible de regrouper des fonctions dédiées à la validation de paramètre au sein d'un module. Le problème est qu'un module définit un état de session différent de la session courante de PowerShell, ce qui fait que dans ce cas la variable **$_** ne peut être utilisée, car elle ne référence pas la valeur de l'argument, mais une autre valeur. Du coup la validation réussit.

On doit donc déclarer un paramètre dans la fonction de validation :

```
$VMException="System.Management.Automation.ValidationMetadataException"
$ErrorMsg= "Le globbing (?,*,[]) n'est pas supporté ({0})."

Function Test-ContainsWildcardCharacters($InputObject){
If ([Management.Automation.WildcardPattern]::ContainsWildcardCharacters(
    $InputObject))
  { throw (new-object $VMException ($ErrorMsg -F $InputObject)) }
 $true }
```

Puis dans le code de la déclaration de l'attribut, on lui passe en ligne de commande la valeur de son argument :

```
function Test( [ValidateScript( {Test-ContainsWildcardCharacters $_ } )]
               [Parameter(ValueFromPipeline = $true)] $Path)
{ $Path }
"C:\*.exe"|Test
```

Ainsi, on obtient la même erreur :

```
Test : Impossible de valider l'argument sur le paramètre « Path ». "Le globbing (?,*,[]) n'est pas supporté (C:\*.exe).
```

Notez que dans ce contexte la variable **$_** est la valeur courante de l'argument, la présence d'un segment de pipeline n'est donc pas nécessaire.

Ce que nous confirme l'exemple suivant :

```
Test "C:\*.exe"
```

Mais il existe une autre approche que Bruce Payette mentionne dans la nouvelle version de son ouvrage, qui est de récupérer la valeur de la variable ***$_*** **dans la portée de l'appelant.**

Elle nécessite toutefois de préciser l'attribut **CmdletBinding**, ainsi qu'une clause *param* vide, afin d'accéder à la variable *$PSCmdlet*. Celle-ci permettant d'accéder aux informations de contexte de l'appelant. En un mot, accéder au contenu de la variable ***$_*** de la fonction utilisant le code de validation :

```
Function Test-ContainsWildcardCharacters{
  #Nécessaire pour accéder à $PSCmdlet.
 [CmdletBinding()]
 param ()
  $ObjectInScopeOfCaller=$PSCmdlet.SessionState.PSVariable.Get("_").Value
  If
([Management.Automation.WildcardPattern]::ContainsWildcardCharacters($Obje
ctInScopeOfCaller))
  { throw (new-object VMException ($MessageTable.EVAGlobbing -F
$ObjectInScopeOfCaller)) }
 $true
} #Test-ContainsWildcardCharacters
```

Le code s'en trouve simplifié :

```
function Test( [ValidateScript( {Test-ContainsWildcardCharacters} )]
               [Parameter(ValueFromPipeline = $true)] $Path)
{ $Path }
"C:\*.exe"|Test
```

Le revers de la médaille est qu'un appel de notre fonction en dehors du contexte de l'attribut *ValidateScript*, qui renseigne automatiquement la variable *$_*, ne peut pas fonctionner.

Si on souhaite appeler ce type de fonction, on doit renseigner la variable *$_* avant d'appeler notre fonction :

```
Test-ContainsWildcardCharacters "C:\*.exe"
```

```
Impossible de trouver un paramètre positionnel acceptant l'argument « C:\*.exe ».
```

```
"C:\*.exe"|Test-ContainsWildcardCharacters
```

```
L'objet d'entrée ne peut être lié à aucun paramètre de la commande, soit parce que cette commande n'accepte pas l'entrée de pipeline, soit parce que l'entrée et ses propriétés ne correspondent à aucun des paramètres qui acceptent l'entrée de pipeline.
```

```
$_="C:\*.exe"
Test-ContainsWildcardCharacters
```

```
True
```

```
$_="C:\*.exe";Test-ContainsWildcardCharacters
```

```
"Le globbing (?,*,[]) n'est pas supporté (C:\*.exe)."
```

La différence de comportement, entre les deux derniers appels, est due au fait que dans la console on exécute chaque instruction dans un pipeline différent.

Le code suivant fonctionne, car le code d'un scriptblock utilise le même pipeline :

```
&{
$_="C:\*.exe"
Test-ContainsWildcardCharacters
}
```

```
"Le globbing (?,*,[]) n'est pas supporté (C:\*.exe)."
```

A vous de choisir qu'elle est l'approche la plus adaptée à vos besoins.

5.12 Différences de comportement entre une fonction avancée et un cmdlet

Extrait du fichier «about_Functions_Advanced.txt»

« Les fonctions avancées diffèrent des cmdlets compilés pour les raisons suivantes :

- La liaison des paramètres des fonctions avancées ne lève pas d'exception lorsqu'un tableau de chaînes est lié à un paramètre booléen.

- L'attribut ValidateSet et l'attribut ValidatePattern ne peuvent pas passer de paramètres nommés.

- Les fonctions avancées ne peuvent pas être utilisées dans les transactions. »

6 Création de paramètres dynamiques

Une fonction/script avancé peut définir des paramètres accessibles sous certaines conditions, par exemple quand :

- l'argument d'un autre paramètre à une valeur spécifique,
- on exécute le script à partir d'un contexte de provider particulier,
- on dispose de droits supplémentaires, etc.

La plupart des paramètres dynamiques des cmdlets sont ajoutés par des providers lors de l'exécution et sont désignés sous le nom de paramètre dynamique parce qu'ils sont ajoutés seulement quand ils sont nécessaires, c'est-à-dire quand la ou les conditions sont remplies.

Ces conditions valident donc la présence du paramètre dynamique qui renseigne une information contextuelle supplémentaire. Le plus souvent l'usage de paramètre statique, switch et jeu de paramètres suffit.

Un nouveau mot clé a été ajouté au langage, **DynamicParam {<liste-instructions>}**. Il précise le bloc de code définissant le ou les paramètres dynamiques. Sa présence force à déclarer un des blocs *Begin, Process, End*, sans quoi on obtient une erreur.

De plus, la présence d'un attribut **Parameter** ou **CmdletBinding()** est un pré requis à l'exécution du bloc **DynamicParam**. Dans la liste d'instructions, utilisez une instruction *If* pour

spécifier les conditions dans lesquelles le ou les paramètres sont disponibles dans la fonction/script.

La création de notre paramètre nécessite d'utiliser des classes dotnet du Framework PowerShell.

On doit créer un objet :

- *System.Management.Automation.ParameterAttribute* pour représenter un attribut d'un paramètre, tel que **Mandatory**, **Position** ou **ValueFromPipeline**, ou son jeu de paramètres,
- *System.Collections.ObjectModel.Collection[System.Attribute]*, qui est une liste générique hébergeant les attributs d'un paramètre,
- *System.Management.Automation.RuntimeDefinedParameter* pour représenter un paramètre dynamique et spécifier son nom,
- *RuntimeDefinedParameterDictionary* pour retourner au runtime les paramètres construit dynamiquement.

L'exemple suivant, simulant une installation d'un logiciel, crée un paramètre dynamique de type switch nommé *Reboot*.

Si on précise ce switch on redémarrera le poste une fois l'installation terminée, à condition que l'utilisateur de la session possède les droits d'administrateur :

```
function isAdmin {
  $identity = [System.Security.Principal.WindowsIdentity]::GetCurrent()
   $principal = new-object `
                System.Security.Principal.WindowsPrincipal($identity)
   $admin = [System.Security.Principal.WindowsBuiltInRole]::Administrator
  $principal.IsInRole($admin)
}

function Publish-Application{
  [cmdletbinding()]
  Param ([String]$Name)

  DynamicParam
  {
   Write-Warning "Traitement de la clause DynamicParam"
   if (isAdmin)
    {
      Write-Warning "Création du paramètre Reboot "
      $attributes = new-object `
System.Management.Automation.ParameterAttribute
      $attributes.ParameterSetName = 'Admin'
      $attributes.Mandatory = $false
```

```
      $attributeCollection = new-object `
System.Collections.ObjectModel.Collection[System.Attribute]
      $attributeCollection.Add($attributes)
      $dynParam1 = new-object `
System.Management.Automation.RuntimeDefinedParameter(
                     "Reboot",
                     [System.Management.Automation.SwitchParameter],
                     $attributeCollection)

      $paramDictionary = new-object `
System.Management.Automation.RuntimeDefinedParameterDictionary
      $paramDictionary.Add("Reboot", $dynParam1)
      $paramDictionary
    }#if
   #Else : renvoi implicitement $null
  }#DynamicParam

  Begin {
   Write-Host "Traitement du bloc Begin" -fore white
  }#begin

  End{
   $S="Liste des paramètres dynamique : "
   # Write-host "$ $($pscmdlet.GetDynamicParameters().Keys)"
   $PSBoundParameters
   Write-Host "Traitement du bloc End" -fore Green
   Write-Host "Installation de l'application $Name."
   [switch]$Reboot= $null
   if ($PSBoundParameters.TryGetValue('Reboot',[REF]$Reboot) )
   {
    if ($Reboot) #Peut être à false -reboot:false
     {Write-Host "Reboot du poste après traitement." -fore white }
   }
  }#end
}#Publish-Application

Publish-Application -name "OpenOffice" -Reboot
```

Notez que :

- les variables *attributes, attributeCollection, paramDictionary* et *dynParam1* persistent dans la portée courante de la fonction,

- la variable du paramètre *Reboot* n'est pas créée, on doit y accéder par les variables *PSBoundparameter* ou *dynParam1*.

Le résultat pour un compte possédant les droits d'administrateur :

```
PS C:\temp> Publish-Application -name "OpenOffice" -Reboot
AVERTISSEMENT : Traitement de la clause DynamicParam
AVERTISSEMENT : Création du paramètre Reboot
Traitement du bloc Begin

Key                                                    Value
---                                                    -----
Name                                                   OpenOffice
Reboot                                                 True
Traitement du bloc End
Installation de l'application OpenOffice.
Reboot du poste après traitement.
```

Si on ne précise pas le paramètre -*Reboot* :

```
PS C:\temp> Publish-Application -name "OpenOffice"
AVERTISSEMENT : Traitement de la clause DynamicParam
AVERTISSEMENT : Création du paramètre Reboot
Traitement du bloc Begin

Key                                                    Value
---                                                    -----
Name                                                   OpenOffice
Traitement du bloc End
Installation de l'application OpenOffice.
```

On constate que le paramètre dynamique est tout de même construit avant l'opération de liaison.

Si l'utilisateur précise le paramètre –*Reboot*, mais qu'il ne possède pas les droits d'administrateur PowerShell déclenchera l'erreur suivante :

```
PS C:\temp> Publish-Application -name "OpenOffice" -Reboot
AVERTISSEMENT : Traitement de la clause DynamicParam
Publish-Application : Impossible de trouver un paramètre correspondant au nom « Reboot ».
```

Ce qui est normal, car dans ce cas les conditions de création du paramètre ne sont pas remplies. S'il ne précise pas le paramètre –*Reboot*, alors le bloc de création du paramètre dynamique n'est pas exécuté :

```
PS C:\temp> Publish-Application -name "OpenOffice"
AVERTISSEMENT : Traitement de la clause DynamicParam
Traitement du bloc Begin

Key                                                    Value
---                                                    -----
Name                                                   OpenOffice
Traitement du bloc End
Installation de l'application OpenOffice.
```

Si vous dé commentez la ligne suivante :

```
Write-host "Liste des paramètres dynamique : $($pscmdlet.GetDynamicParameters().Keys)"
```

Un affichage supplémentaire du message "Traitement de la clause DynamicParam" aura lieu, il est dû à l'appel *$PSCmdlet.GetDynamicParameters()*. Le bloc ***DynamicParam*** est donc exécuté à chaque appel de la méthode *GetDynamicParameters()*, par conséquent ce bloc ne doit contenir que du code spécifique à la création de paramètre dynamique, ni plus ni moins.

Étant donné que l'appel de **Get-Command** construit les métadonnées de l'objet précisé, ce cmdlet appelle en interne la méthode *GetDynamicParameters()* afin de retrouver tous les paramètres de l'objet précisé :

```
(gcm Publish-Application).Parameters.Values|Where {$_.IsDynamic}|Select
Name
```

Il reste que l'exemple mis en œuvre dans la fonction *Publish-Application* met en évidence le problème suivant : vous pouvez utiliser un paramètre dynamique sans tenir compte des conditions de sa création, mais dans ce cas, comme on l'a vu, le script est susceptible de déclencher une exception. Pour la gérer soit on utilisera le paramètre *–ErrorAction*, soit on testera une seconde fois la condition (isAdmin) pour éviter cette exception.

Voyons un autre exemple, celui indiqué dans la documentation de PowerShell :

```
function Sample {
 [cmdletbinding()]
  Param (
   [String]$Name,
   [String]$path)

  DynamicParam
  {
   if ($path -match "^HKLM:")
    {
      $attributes = new-object `
                      System.Management.Automation.ParameterAttribute
      $attributes.ParameterSetName = 'pset1'
      $attributes.Mandatory = $false
      $attributeCollection = new-object `
             System.Collections.ObjectModel.Collection[System.Attribute]
      $attributeCollection.Add($attributes)
      $dynParam1 = new-object `
                    System.Management.Automation.RuntimeDefinedParameter(
                         "dp1",
                         [Int32],
                         $attributeCollection)

      $paramDictionary = new-object `
           System.Management.Automation.RuntimeDefinedParameterDictionary
      $paramDictionary.Add("dp1", $dynParam1)
      $paramDictionary
    }
  }
```

```
  end{
   [Int]$Dp1= $null
   if ($PSBoundParameters.TryGetValue('Dp1',[REF]$Dp1) )
   {
    if ($Dp1)
     {Write-Host "Valeur du paramètre dynamique `$dp1=$Dp1" –fore white }
   }
  }
}#Sample
```

Pour ce cas on reconnaît, dans l'appel de la fonction, la présence de la condition à l'aide du contenu du paramètre *Path*, il référence HKLM :

```
Sample -name "test" -path "hklm:\" -dp1 5
```

Lorsqu'on lira ce code, on est assuré qu'il ne déclenchera pas d'exception, sous réserve de connaître l'existence du paramètre dynamique et de son fonctionnement.

Un autre exemple basé sur une réécriture :

```
function Sample {
 [cmdletbinding()]
  Param ([String]$Name)
  DynamicParam
  {
     if ((Get-Location).Provider.Name –eq "FileSystem")
    {
     # Le reste du code est identique à celui de l'exemple précédent
     …
}#Sample
```

Cette fois-ci ce n'est plus un paramètre de la fonction qui détermine la condition, mais la localisation courante. Là aussi la condition est présente dans le code, *cd C:\Windows* :

```
cd C:\Windows
Sample -name "test" -dp1 5
```

Enfin, le bloc ***DynamicParam*** ne doit renvoyer dans le pipeline que des paramètres dynamiques, sinon on obtient l'erreur suivante :

```
Sample : Impossible de récupérer les paramètres dynamiques pour l'applet de commande. Le type à retourner du bloc dynamicparam pour l'objet « System.Object[] » est incorrect. Le bloc dynamicparam doit retourner soit $null, soit un objet de type [System.Management.Automation.RuntimeDefinedParameterDictionary].
```

Voir aussi :

Make a parameter mandatory only if another parameter value is provided

http://www.powergui.org/thread.jspa?threadID=10607

Exemple C# basé sur la présence d'un switch

http://msdn.microsoft.com/en-us/library/dd878334(VS.85).aspx

7 La variable automatique PSCmdlet

Comme nous l'avons déjà vu, la présence dans une fonction ou un script d'un attribut **Parameter** ou **CmdletBinding** fait que PowerShell déclare une variable automatique nommée *$PSCmdlet*.

C'est une instance de la classe interne **PSScriptCmdlet** qui hérite de la classe **PSCmdlet**, une des classes utilisées pour créer un cmdlet. Cette variable permet d'accéder, au travers de ces méthodes, à certaines fonctionnalités du moteur PowerShell.

```
Function FcntAvancée{
  [CmdletBinding()]
  Param (
  [Parameter(
      ValueFromPipeline = $true)]
  $NomParam,
  $Count)
   Write-host "`$NomParam=$NomParam `t `$Count=$Count"
   $pscmdlet|gm |Sort membertype,name
}

FcntAvancée un 9
```

Elle contient les membres publics suivants :

```
Name                                MemberType
----                                ----------
CommandOrigin                       Property
CommandRuntime                      Property
CurrentPSTransaction                Property
Events                              Property
Host                                Property
InvokeCommand                       Property
InvokeProvider                      Property
JobRepository                       Property
MyInvocation                        Property
ParameterSetName                    Property
SessionState                        Property
Stopping                            Property

CurrentProviderLocation             Method
Dispose                             Method
Equals                              Method
GetDynamicParameters                Method
GetHashCode                         Method
GetResolvedProviderPathFromPSPath   Method
GetResourceString                   Method
GetType                             Method
GetUnresolvedProviderPathFromPSPath Method
GetVariableValue                    Method
Invoke                              Method
```

ShouldContinue	Method
ShouldProcess	Method
ThrowTerminatingError	Method
ToString	Method
TransactionAvailable	Method
WriteCommandDetail	Method
WriteDebug	Method
WriteError	Method
WriteObject	Method
WriteProgress	Method
WriteVerbose	Method
WriteWarning	Method

Malheureusement le SDK en ligne ne propose pas de documentation sur cette classe, mais vous pouvez consulter celle de la classe ***Cmdlet***

Elle permet de :

- déterminer le jeu de paramètres courant (*ParameterSetName*), s'il y en a un,
- d'écrire directement dans les différents pipelines (WriteError, etc),
- de gérer les paramètres -Confirm et -Whatif,
- de retrouver le chemin courant d'un provider (*CurrentProviderLocation*),
- déterminer si la fonction/script peut participer à une transaction (*TransactionAvailable*),
- retrouver les paramètres dynamiques (*GetDynamicParameters*),
- d'extraire une ressource des assembly de cmdlets localisés (*GetResourceString*), etc.

La méthode *WriteCommandDetail* permet de tracer le détail d'exécution du pipeline dans l'eventlog PowerShell, vous devez toutefois configurer les snapins de la manière suivante :

```
Get-PSSnapin|
 Foreach {
  Write-Warning "Trace le détail d'exécution du pipeline pour le snapin :`
$_.Name";
  $_.LogPipelineExecutionDetails=$true
 }
```

7.1 Gestion des exceptions

La variable *$PSCmdlet* permet également de préciser les informations liées à une exception :

```
Function FcntAvancée{
  [CmdletBinding()]
  Param (
    [Parameter(ValueFromPipeline = $true)]
   $NomParam,
   $Count)
  Begin {
      $pscmdlet.WriteDebug("**** Test de log --------")
```

```
    }
 Process {
     #Throw "Erreur: l'objet n'existe pas."
     $ErrorRecord = New-Object System.Management.Automation.ErrorRecord  (
         (New-Object Exception "Erreur: l'objet n'existe pas."),
            "FcntAvancée.MonErreur_1",
            [System.Management.Automation.ErrorCategory]::ObjectNotFound,
            $NomParam
         )

      #$ErrorRecord.ErrorDetails="Informations supplémentaires: actions
recommandées (Indiquer un objet existant)"
      $PSCmdlet.ThrowTerminatingError($ErrorRecord)
   }
 end{
   Write-host "`$NomParam=$NomParam `t `$Count=$Count"
   $pscmdlet|gm |Sort membertype,name
 }
}

FcntAvancée un 9
```

Utilisons l'instruction **Throw($Message)** :

```
Erreur: l'objet n'existe pas.
Au niveau de ligne : 22 Caractère : 12
+       Throw <<<<  "Erreur: l'objet n'existe pas."
    + CategoryInfo          : OperationStopped: (Erreur: l'objet n'existe pas.:String) [], RuntimeException
    + FullyQualifiedErrorId : Erreur: l'objet n'existe pas.
```

Avec ***Resolve-Error*** on connaît la ligne exacte, mais les informations concernant le pipeline ne sont pas renseignées ni celles des paramètres liés :

```
Exception             : System.Management.Automation.RuntimeException: Erreur: l'objet n'existe pas.
TargetObject          : Erreur: l'objet n'existe pas.
CategoryInfo            : OperationStopped: (Erreur: l'objet n'existe pas.:String) [], RuntimeException
FullyQualifiedErrorId : Erreur: l'objet n'existe pas.
```

Si on utilise *$PSCmdlet.ThrowTerminatingError($ErrorRecord)*

```
FcntAvancée : Erreur: l'objet n'existe pas.
Au niveau de ligne : 1 Caractère : 12
+ FcntAvancée <<<<  un 9
    + CategoryInfo          : ObjectNotFound: (un:String) [FcntAvancée], Exception
    + FullyQualifiedErrorId : FcntAvancée.MonErreur_1,FcntAvancée
```

On ne connaît plus le numéro de ligne, mais on peut retrouver l'id de l'erreur précisé dans le code : *FcntAvancée.MonErreur_1*.

Avec ***Resolve-Error*** les informations concernant le pipeline sont renseignées ainsi que celles des paramètres liés :

```
Exception             : System.Exception: Erreur: l'objet n'existe pas.
```

```
                          à System.Management.Automation.MshCommandRuntime.ThrowTerminatingError(
                          ErrorRecord errorRecord)
TargetObject            : un
CategoryInfo            : ObjectNotFound: (un:String) [FcntAvancée], Exception
FullyQualifiedErrorId : FcntAvancée.MonErreur_1,FcntAvancée
```

De plus, le champ *CommandOrigin* contient la valeur Runspace au lieu d'Internal lorsqu'on utilise l'instruction **Throw**.

Renseigner le champ *ErrorDetails* informe l'utilisateur, de votre fonction/script, des possibles actions corrigeant l'erreur.

Sans :

```
FcntAvancée : Erreur : l'objet n'existe pas.
```

Avec :

```
FcntAvancée : Informations supplémentaires : actions recommandées (Indiquer un objet existant)
```

Cette approche est plus explicite pour l'utilisateur, le message de l'exception restant accessible via :

```
$error[0].exception
Erreur: l'objet n'existe pas.
```

7.2 La variable ErrorView

Avec **Throw** :

```
$ErrorView="CategoryView"
FcntAvancée un 9
OperationStopped: (Erreur: l'objet n'existe pas.:String) [], RuntimeException
```

Avec **ThrowTerminatingError** :

```
FcntAvancée un 9
ObjectNotFound: (un:String) [FcntAvancée], Exception
$ErrorView=$null #Affichage normal
```

Vous pouvez utiliser *$PSCmdle.WriteError()* pour les erreurs non bloquantes.

Voir aussi, *Windows PowerShell Error Records* :

http://msdn.microsoft.com/en-us/library/ms714465(VS.85).aspx

Module Write Error: http://poshcode.org/1574

7.3 À propos du bloc End

On peut utiliser le bloc *End* pour libérer des objets alloués dans d'autres blocs, ce bloc est appelé une fois tous les objets du pipeline traités. Mais dans le cas où le mécanisme du pipeline est interrompu par une exception ou par Control-C, ce bloc ne sera pas appelé, car il n'y a pas de propagation de l'événement à chaque segment de pipeline.

En revanche vous pouvez implémenter une gestion des exceptions déclenchées dans votre fonction/script autour de l'exception **PipelineStoppedException :**

```
Process
{
   #Gestion de l'erreur PipelineStoppedException
 trap [System.Management.Automation.PipelineStoppedException]
 {
   $Msg="Stop le pipeline via Break."
   write-Debug ("[{0}]Process: {1}" –F $MyInvocation.InvocationName,$Msg))
    #Finalise les ressources…
   Break
  }
...
```

Ce code gérera les exceptions, mais pas l'arrêt via Control-C.

Soyez donc attentif à ce point si vous comptez utiliser des ressources qui ne sont pas gérées par le Framework dotnet ou qui nécessiteraient un appel explicite à la méthode *Dispose()*.

8 Les arguments de l'attribut CmdletBinding

L'attribut **[CmdletBinding()]** offre la possibilité d'implémenter les paramètres d'atténuation de risques qui ne sont pas implémentés par défaut, à savoir -*WhatIf* et -*Confirm*

Pour rappel :

-*WhatIf* affiche un message décrivant l'effet de la commande, au lieu d'exécuter cette dernière, il permet de simuler l'exécution de la commande.

-*Confirm* invite l'utilisateur à confirmer ou non l'exécution de la commande, et ce, pour chaque objet qu'elle traite.

8.1 SupportsShouldProcess

Lorsque l'argument **SupportsShouldProcess** est défini sur la valeur *$true*, il indique que la fonction avancée prend en charge les appels à la méthode *ShouldProcess* qui est utilisée pour demander la confirmation de l'utilisateur avant que la fonction n'exécute une action qui modifierait le système. La fonction avancée peut continuer selon la valeur booléenne retournée par la méthode *ShouldProcess*.

Lorsque cet argument est spécifié, les paramètres -*Confirm* et -*WhatIf* sont activés pour la fonction avancée.

Note : si l'utilisateur spécifie le paramètre -*verbose*, il sera avisé de l'opération même s'il n'y a pas de demande de confirmation.

```
Function TestAttenuationRisque1
{ [CmdletBinding(SupportsShouldProcess=$True)]
   param(
    [Parameter(Position=0, Mandatory = $true,
               ValueFromPipeline = $true)]$ID)
```

```
  Begin
  { Function Traitement($Object) {Write-Host "Traite $Object"} }
  Process
  {
   if ($psCmdlet.shouldProcess("Opération Traitement"))
    { Traitement $_}
   else {Write-host "Pas de traitement avec ShouldProcess"}
  }#Process
}
```

Déclenchons l'appel de la méthode *ShouldProcess* sans effectuer le traitement :

```
1..2|TestAttenuationRisque1 -whatif
```

```
WhatIf : Opération « TestAttenuationRisque1 » en cours sur la cible « Opération Traitement ».
Pas de traitement avec ShouldProcess
WhatIf : Opération « TestAttenuationRisque1 » en cours sur la cible « Opération Traitement ».
Pas de traitement avec ShouldProcess
```

Déclenchons l'appel de la méthode *ShouldProcess* en demandant une confirmation :

```
1..2|TestAttenuationRisque1 -confirm
```

Voici une recopie d'écran des différents comportements selon les réponses données lors des demandes de confirmation :

```
PS C:\temp> 1..2|TestAttenuationRisque1 -confirm

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « TestAttenuationRisque1 » en cours sur la cible « Opération Traitement ».
[O] Oui  [T] Oui pour tout  [N] Non  [U] Non pour tout  [S] Suspendre  [?] Aide (la valeur par défaut est « O ») : O
Traite 1

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « TestAttenuationRisque1 » en cours sur la cible « Opération Traitement ».
[O] Oui  [T] Oui pour tout  [N] Non  [U] Non pour tout  [S] Suspendre  [?] Aide (la valeur par défaut est « O ») : O
Traite 2
PS C:\temp> 1..2|TestAttenuationRisque1 -confirm

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « TestAttenuationRisque1 » en cours sur la cible « Opération Traitement ».
[O] Oui  [T] Oui pour tout  [N] Non  [U] Non pour tout  [S] Suspendre  [?] Aide (la valeur par défaut est « O ») : T
Traite 1
Traite 2
PS C:\temp> 1..2|TestAttenuationRisque1 -confirm

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « TestAttenuationRisque1 » en cours sur la cible « Opération Traitement ».
[O] Oui  [T] Oui pour tout  [N] Non  [U] Non pour tout  [S] Suspendre  [?] Aide (la valeur par défaut est « O ») : U
Pas de traitement avec ShouldProcess
Pas de traitement avec ShouldProcess
PS C:\temp> 1..2|TestAttenuationRisque1 -confirm

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « TestAttenuationRisque1 » en cours sur la cible « Opération Traitement ».
[O] Oui  [T] Oui pour tout  [N] Non  [U] Non pour tout  [S] Suspendre  [?] Aide (la valeur par défaut est « O ») : N
Pas de traitement avec ShouldProcess

Confirmer
Êtes-vous sûr de vouloir effectuer cette action ?
Opération « TestAttenuationRisque1 » en cours sur la cible « Opération Traitement ».
[O] Oui  [T] Oui pour tout  [N] Non  [U] Non pour tout  [S] Suspendre  [?] Aide (la valeur par défaut est « O ») : O
Traite 2
```

On a le choix de confirmer tous les traitements ou aucun ou de les sélectionner un par un.

8.1.1 À propos des capacités des providers

Les capacités de provider sont des caractéristiques implémentées ou supportées par le provider. Plusieurs peuvent être spécifiées, voire une seule. Pour connaître les capacités d'un provider, on utilise le cmdlet **Get-PsProvider :**

```
Get-PSProvider
```

```
Name            Capabilities                Drives
----            ------------                ------
WSMan           Credentials                 {WSMan}
FileSystem      Filter, ShouldProcess       {C, D }
PscxSettings    None                        {Pscx}
…
```

Ici le provider PSCX ou WSMan n'implémente pas la capacité ShouldProcess, ce qui signifie que sur ces providers l'utilisation des paramètres *-WhatIf* et *-Confirm* est inopérant :

```
cd pscx:
dir w*
 #L'appel à Remove-Item n'a aucun effet.
remove-item w* -whatif # ou –Confirm
dir w*
```

De plus, un provider peut ne pas implémenter la gestion de certains cmdlets de provider (http://msdn.microsoft.com/en-us/library/ee126197(VS.85).aspx), si c'est le cas l'exécution d'un tel cmdlet peut n'avoir aucun effet ou déclencher une exception de type **InvalidOperationException** :

```
cd wsman:\localhost
Dir
Remove-Item maxtimeoutms -confirm
```

```
Remove-Item : Impossible d'utiliser cette commande dans le chemin d'accès actuel, car Remove-Item n'est pas
pris en charge à ce niveau du chemin d'accès de fournisseur.
```

Il est possible de déterminer quelles capacités le provider courant implémente :

```
(get-location).provider.capabilities
```

Voici la liste des cmdlets implémentant le paramètre *–Whatif* :

```
Get-Help * -Parameter Whatif
```

Note : le code de la fonction avancée doit, dans la section d'aide, déclarer une entrée pour le paramètre *–Whatif*. Sinon votre fonction ne sera pas listée.

Voir aussi le détail des capacités des providers :

http://msdn.microsoft.com/en-us/library/system.management.automation.provider.providercapabilities(VS.85).aspx

8.1.2 La méthode ShouldContinue

La variable automatique nommée *$PSCmdlet* propose également la méthode *ShouldContinue*.

Elle est utilisée pour demander un second message de confirmation. Elle doit être appelée lorsque la méthode *ShouldProcess* retourne *$true.* Cette méthode n'est pas affectée par le paramétrage des préférences ou par un paramètre.

Modifions le bloc *process* de la fonction précédente :

```
  Process
  {
    if ($psCmdlet.shouldContinue("Requête ?", "Caption"))
      { Traitement $_}
```

```
    else {Write-host "Pas de traitement avec ShouldContinue"}
  }#Process
}
1..3|TestAttenuationRisque1
```

Ainsi codé, la présence ou non de -*Confirm* ou -*Whatif* n'influence pas l'exécution du code, la demande de confirmation est toujours déclenchée.

Pour mettre en place une double confirmation, il faut combiner les deux appels :

Pour plus d'informations sur la façon de demander confirmation, consultez " Requesting Confirmation " sur MSDN : http://go.microsoft.com/fwlink/?LinkID=136658

Note :

Une fonction avancée appelant *ShouldContinue* devrait également mettre en application un paramètre –*Force*, de type [switch], qui permettrait au code de la fonction avancée d'outrepasser cette demande de confirmation, et ainsi poursuivre l'opération. Assurez-vous que son implémentation n'affecte pas les appels à *ShouldProcess*.

Un exemple d'implémentation :

```
Function TestAttenuationRisque1_2
{
  [CmdletBinding(SupportsShouldProcess=$True)]
   param(
    [Parameter(Position=0,  Mandatory = $true,
               ValueFromPipeline = $true)] $ID,
    [Switch]$Force)
  Begin
  { Function Traitement($Object) {Write-Host "Traite $Object"} }

  Process
  {
   if ($psCmdlet.shouldProcess($_, "Opération Traitement"))
    {
      if ($force –or $pscmdlet.ShouldContinue($_, "Opération Traitement"))
       { Traitement $_}
    }
   else {Write-host "Pas de traitement avec SouldProcess"}
  }#Process

}
```

```
1..3|TestAttenuationRisque1_2 -whatif
1..3|TestAttenuationRisque1_2 -confirm
```

8.2 ConfirmImpact

Certaines opérations modifiant le système sont jugées à risques, telles que le reformatage d'une partition active du disque dur. Pour ces cas, le cmdlet ou la fonction avancée devrait fixer la valeur de **ConfirmImpact** à *High*. Cet argument spécifie à quel moment l'action de la fonction doit être confirmée lors de la méthode *ShouldProcess*.

L'appel à la méthode *ShouldProcess* affiche une demande de confirmation uniquement lorsque l'argument **ConfirmImpact** est égal ou supérieur à la valeur de la variable de préférence *$ConfirmPreference*.

Spécifiez cet argument uniquement lorsque l'argument **SupportsShouldProcess** est également spécifié. La valeur par défaut de cet argument est *Medium*.

Cet argument force le cmdlet à demander une confirmation de l'utilisateur même lorsque celui-ci n'a pas précisé le paramètre *-Confirm*. Évitez toutefois une utilisation excessive de la valeur **ConfirmImpact** à *High*, n'oubliez pas qu'à l'origine PS est un outil d'automatisation de tâches.

8.2.1 La variable $ConfirmPreference

Cette variable automatique agit sur le comportement des cmdlets/fonctions avancées utilisant l'argument **ConfirmImpact.** Lorsque la valeur de ***$ConfirmPreference*** [*High* (par défaut), *Medium, Low, None*] est inférieure ou égale au niveau de risque de l'action du cmdlet, PowerShell demande automatiquement la confirmation de l'utilisateur avant d'exécuter l'action. Mais si sa valeur est supérieure au niveau du risque de l'action du cmdlet, PowerShell exécutera l'action sans demander de confirmation.

Dans ce cas, l'utilisateur devra préciser le paramètre *–Confirm* pour modifier le comportement induit par la valeur de la variable ***$ConfirmPreference***.

Valeur	Description
High	Les actions de cmdlet avec un risque élevé sont automatiquement confirmées. Pour activer la confirmation d'une commande spécifique, utilisez *-Confirm*. Pour supprimer la confirmation d'une commande spécifique, utilisez *-Confirm:$false*
Medium	Les actions de cmdlet avec un risque moyen ou élevé sont automatiquement confirmées. Pour activer la confirmation d'une commande spécifique, utilisez *-confirm*. Pour supprimer la confirmation d'une commande spécifique, utilisez *-Confirm:$false*.

Low	Les actions de cmdlet avec un risque faible, moyen ou élevé sont automatiquement confirmées. Pour supprimer la confirmation d'une commande spécifique, utilisez *-Confirm:$false*.
None	Aucune action de cmdlet n'est confirmée automatiquement. Les utilisateurs doivent utiliser le paramètre *-Confirm* pour demander la confirmation de commandes spécifiques.

Pour résumer, l'argument **ConfirmImpact** configure le niveau de confirmation de votre fonction avancée et la variable automatique *$ConfirmPreference* indique quels sont les niveaux qui nécessiteront une confirmation :

ConfirmImpact	$ConfirmPreference	Action
Medium	High	Pas de demande de confirmation automatique.
Low	Medium	Pas de demande de confirmation automatique.
None	*	Aucune demande de confirmation. Même si on précise *–Confirm*.
*	None	Seule la présence de *–Confirm* déclenchera une demande de confirmation.
High	High	Demande de confirmation automatique.
Medium	Low	Demande de confirmation automatique.

Dans tous les cas l'usage du paramètre *–Confirm* reste possible, sauf si l'argument **ConfirmImpact** est à *None.*

8.2.2 La variable $WhatIfPreference

Elle détermine si le paramètre *-WhatIf* est activé automatiquement pour chaque commande qui le prend en charge. Ses valeurs possibles sont :

0 (par défaut)	Le paramètre *-WhatIf* n'est pas activé automatiquement. Pour l'activer manuellement, utilisez le paramètre *-WhatIf* de la commande.
1	Le paramètre *-WhatIf* est activé automatiquement sur toute commande qui le prend en charge. Pour le désactiver manuellement utilisez : *-WhatIf:$false*

8.3 SupportsTransactions

Bien que la présence de cet attribut soit correctement interprétée, il n'est apparemment ni documenté ni supporté par Microsoft, peut-être dans la prochaine version.

Notez que les proxys de commande génèrent du code utilisant cet attribut…

Sa valeur de type booléen indique si la fonction accepte ou non un rollback de transaction PowerShell. Si elle vaut $true le paramètre *UseTransaction* est alors ajouté à la liste des paramètres de la fonction/script.

Voir aussi *$PScmdlet.TransactionAvailable()*

8.4 DefaultParameterSetName

Comme nous l'avons précédemment vu, l'argument *DefaultParameterSetName* spécifie le nom du jeu de paramètres que Windows PowerShell essaiera d'utiliser lorsqu'il ne pourra pas déterminer quel jeu de paramètres utiliser. Vous pouvez éviter ce problème en faisant du paramètre unique de chaque jeu de paramètres un paramètre obligatoire :

```
[CmdletBinding(DefaultParameterSetName="MaValeurParDefaut")]
```

Le nom attribué à *DefaultParameterSetName* peut être arbitraire ou reprendre un de ceux utilisés dans la liste des paramètres.

9 Accéder aux détails d'une commande

Le cdmlet **Get-Member** utilise le système de réflexion de dotnet afin de renvoyer les membres d'un objet ou d'une classe, ce cmdlet renvoie un objet, une donnée, détaillant un objet ou une classe.

Nous avons vu qu'il est possible de placer sur un paramètre de fonction des métadonnées à l'aide d'attributs, ces informations sont destinées au compilateur interne de PowerShell. Si on le souhaite, on peut les manipuler une fois la fonction compilée.

9.1 Accéder aux informations d'une fonction

Cette nouvelle version apporte une nouveauté qui est de retrouver les métadonnées d'une commande :

```
Function FcntAvancée{
  Param (
  [Parameter(ValueFromPipeline = $true)]
  [ValidateNotNullOrEmpty()]
  $NomParam,
  $Count)
  Write-host "`$NomParam=$NomParam `t `$Count=$Count"
}
```

Le cmdlet **Get-Member** renvoi uniquement la liste des membres de la classe ***FunctionInfo*** en tant que tableau d'instances de type ***MemberDefinition*** :

```
$Fi=Get-Member -input (Get-Item Function:fcntavancée)
$Fi.Parameters # Parameters ne contient aucune donnée
```

Pour récupérer les métadonnées de notre fonction, on doit utiliser le cmdlet **Get-Item** :

```
$Fi=Get-Item Function:fcntavancée
```

Cette fois on récupère une instance de la classe ***FunctionInfo***.

Huit nouvelles propriétés y ont été ajoutées et permettent d'accéder aux métadonnées définies par les attributs :

Name	Description
CmdletBinding	Gets a Boolean value that indicates whether the function uses the same parameter binding that compiled cmdlets use. This property is introduced in Windows PowerShell 2.0.
CommandType	Gets the Windows PowerShell defined type of the command, such as a cmdlet, function, filter, or script. (Inherited from CommandInfo)
DefaultParameterSet	Gets the name of the parameter set that is used if Windows PowerShell cannot determine which parameter set to use when the function is run. This property is introduced in Windows PowerShell 2.0.
Definition	Overridden. Gets the string definition of the function.
Description	Gets and sets a description of the function. This property is introduced in Windows PowerShell 2.0.
Module	Gets information about the module that defines this command. This property is introduced in Windows PowerShell 2.0. (Inherited from CommandInfo)
ModuleName	Gets the name of the module that defines the command. This property is introduced in Windows PowerShell 2.0. (Inherited from CommandInfo)
Name	Gets the name of the command. (Inherited from CommandInfo)
Options	Gets and sets the scope options for the function.
OutputType	Overridden. Gets the .NET Framework types returned by the function.
Parameters	Gets the parameters of the command. This property is introduced in Windows PowerShell 2.0. (Inherited from CommandInfo)
ParameterSets	Gets information about the parameter sets associated with the command. This property is introduced in Windows PowerShell 2.0. (Inherited from CommandInfo)
ScriptBlock	Gets the implementation of the function.
Visibility	Gets and sets a constant that identifies whether the *element* is visible outside the runspace. For example, an element can be a private member of a module and therefore not visible to commands, such as pipeline commands, that are coming from outside the runspace. This property is introduced in Windows PowerShell 2.0. (Inherited from CommandInfo)

Deux propriétés nous intéressent, ***Parameters*** et ***ParameterSets*** :

```
$Fi.Parameters
```

```
Key                  Value
---                  -----
NomParam              System.Management.Automation.ParameterMetadata
Count                 System.Management.Automation.ParameterMetadata
Verbose              *System.Management.Automation.ParameterMetadata
Debug                *System.Management.Automation.ParameterMetadata
ErrorAction          *System.Management.Automation.ParameterMetadata
WarningAction         *System.Management.Automation.ParameterMetadata
ErrorVariable        *System.Management.Automation.ParameterMetadata
WarningVariable       *System.Management.Automation.ParameterMetadata
OutVariable           *System.Management.Automation.ParameterMetadata
OutBuffer            *System.Management.Automation.ParameterMetadata
```

Parameters renvoie une liste des paramètres de la commande, les paramètres communs sont précédés d'une étoile et sont déclarés automatiquement comme pour n'importe quel cmdlet. Leur redéclaration dans la liste des paramètres d'une fonction/script avancé provoquera une erreur :

```
Un paramètre nommé « ErrorAction » a été défini plusieurs fois pour la commande.
```

Les paramètres d'atténuation de risques *-WhatIf* et *-Confirm* ne sont pas listés, car la fonction déclare implicitement :

```
[CmdletBinding(SupportsShouldProcess=$False)]
```

Si vous déclarez l'argument *SupportsTransactions* :

```
[CmdletBinding(SupportsTransactions=$True)]
```

Le paramètre *–UseTransaction*, de type switch, est alors ajouté par PowerShell :

```
Parameter Name: UseTransaction
  ParameterType = System.Management.Automation.SwitchParameter
  Position = -2147483648
  IsMandatory = False
  IsDynamic = False
  HelpMessage =
  ValueFromPipeline = False
  ValueFromPipelineByPropertyName = False
  ValueFromRemainingArguments = False
  Aliases = {usetx}
  Attributes =
    System.Management.Automation.ParameterAttribute …
```

PowerShell version 2 autorise également le paramètre "-?" qui obtient la rubrique d'aide de la commande, si celle-ci en contient une.

On pourrait donc utiliser le paramètre *-ErrorAction* pour masquer un des problèmes d'un cas d'étude précédent :

```
"Un","Deux"|FcntAvancée -Nom "Trois" -c 9 –ErrorAction SilentlyContinue
```

```
$NomParam=Trois       $Count=9
```

Ainsi les erreurs non-bloquantes ne sont plus générées, mais ce n'est pas vraiment une solution.

La valeur du paramètre commun *–ErrorAction* prime sur celle de *$ErrorActionPreference*, que ce soit en utilisant le cmdlet **Write-Error** ou la méthode *$PSCmdlet.WriteError()*.

Il est possible d'obtenir le détail d'un paramètre :

```
$Fi.Parameters.NomParam
```

```
Name            : NomParam
ParameterType   : System.Object
ParameterSets   : {[__AllParameterSets, System.Management.Automation.ParameterSetMetadata]}
IsDynamic       : False
Aliases         : {}
Attributes      : {System.Management.Automation.ValidateNotNullOrEmptyAttribute, __AllParameterSets}
SwitchParameter : False
```

L'objet renvoyé par *NomParam* est de la classe ***ParameterMetadata***, on peut aussi obtenir le détail d'un de ces attributs :

```
$Fi.Parameters.NomParam.Attributes[1]
```

```
Position                        : 0
ParameterSetName                : __AllParameterSets
```

```
Mandatory                       : False
ValueFromPipeline               : True
ValueFromPipelineByPropertyName : False
ValueFromRemainingArguments     : False
HelpMessage                     :
HelpMessageBaseName             :
HelpMessageResourceId           :
TypeId                          : System.Management.Automation.ParameterAttribute
```

Les informations renvoyées différeront selon le type de l'attribut interrogé.

La propriété ***ParameterSets*** héberge le même type d'informations sur les paramètres, mais ceux-ci sont regroupés par jeux de paramètres :

```
$Fi.ParameterSets
```

```
Parameter Set Name: __AllParameterSets
Is default parameter set: False

  Parameter Name: NomParam
    ParameterType = System.Object
    Position = 0
    IsMandatory = False
    IsDynamic = False
    HelpMessage =
    ValueFromPipeline = True
    ValueFromPipelineByPropertyName = False
    ValueFromRemainingArguments = False
    Aliases = {}
    Attributes =
      System.Management.Automation.ValidateNotNullOrEmptyAttribute
      System.Management.Automation.ParameterAttribute

  Parameter Name: Count
    ParameterType = System.Object
    Position = 1
…
```

Si vous utilisez un des précédents exemples utilisant l'attribut ***ParameterSetName***, vous pourrez accéder à chaque jeu de paramètre de la manière suivante :

```
$Fi.ParameterSets.Count
$Fi.ParameterSets.Item(0)
$Fi.ParameterSets.Item(0)|Where {$_.name –eq "LogName "}
```

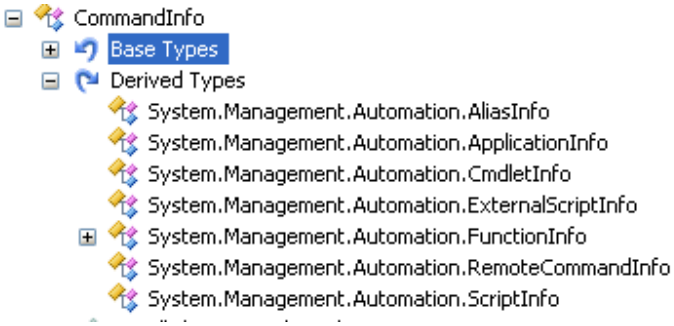
Ces informations pourront, par exemple, nous aider à construire une partie de la documentation d'une fonction avancée ou générer automatiquement une partie d'un jeu de tests.

Nous avons pu obtenir des métadonnées d'une fonction à l'aide du provider de fonction, mais comment obtenir des informations sur un cmdlet compilé ?

9.2 Accéder aux Informations d'un cmdlet

PowerShell n'utilisant pas de provider pour les cmdlets, **Get-Item** ne peut fonctionner. C'est pour cette raison qu'il vaut mieux dans tous les cas utiliser **Get-Command** qui renvoie lui aussi les informations d'un cmdlet. La version 2 permet de récupérer des informations sur les

fonctions, la différence est qu'un cmdlet contient des informations supplémentaires, la classe de l'objet renvoyé sera donc appropriée à la commande récupérée :



Par exemple, un cmdlet contient un champ *PSSnapIn* de type ***PSSnapInInfo***, un script externe .ps1 contient un champ *Path*, ou une application un champ *FileVersionInfo*.

Sachez que l'objet récupéré par **Get-Command** peut être exécuté à l'aide de l'opérateur **& :**

```
$Fi=Get-Command -Type Function fcntavancée
&Fi
$App=Get-Command -Type Application notepad.exe|select –unique
&$App
$Ci=Get-Command  -Type Cmdlet Get-Item
&$Ci
```

9.3 Accéder aux informations d'un fichier script

Les scripts peuvent également utiliser les mêmes fonctionnalités avancées qu'une fonction, comme le montre l'exemple suivant :

http://blogs.msdn.com/powershell/archive/2008/12/23/advanced-functions-and-test-leapyear-ps1.aspx

On utilisera également **Get-Command** :

```
$Si= Get-Command .\Test-LeapYear.ps1
#$Si= Gcm "$pwd\Test-LeapYear.ps1"
```

Cette fois on récupère une instance de la classe ***ExternalScriptInfo***.

9.4 Accéder aux informations d'un module

Il est possible d'obtenir les informations d'un module PowerShell, mais étant donné qu'un module peut héberger plusieurs commandes utilisant soit du code natif soit du code compilé, les classes des instances renvoyées par **Get-Command** ne seront pas toujours les mêmes, mais elles dériveront toutes de ***CommandInfo***.

```
Import-Module BitsTransfer
$Tcmd=Get-Command -Module BitsTransfer ; $Tcmd[0]|Get-Member
```

```
…
Module          Property      System.Management.Automation.PSModuleInfo Module {get;}
…
```

```
$M=Get-Module BitsTransfer
```

```
$M|Get-Member
```

Get-Module renvoie une instance de la classe ***PSModuleInfo***, c'est elle qui contient les métadonnées du module :

```
   TypeName: System.Management.Automation.PSModuleInfo

Name                 MemberType   Definition
----                 ----------    ----------
AsCustomObject       Method       psobject AsCustomObject()
…
ExportedAliases      Property     System.Collections.Generic.Dictionary`2[[System.String,
ExportedCmdlets      Property      System.Collections.Generic.Dictionary`2[[System.String,
ExportedFormatFiles  Property      System.Collections.ObjectModel.ReadOnlyCollection`1[[System.String,
ExportedFunctions    Property     System.Collections.Generic.Dictionary`2[[System.String,
ExportedTypeFiles    Property     System.Collections.ObjectModel.ReadOnlyCollection`1[[System.String,
ExportedVariables    Property      System.Collections.Generic.Dictionary`2[[System.String,
Guid                 Property      System.Guid Guid {get;}
…
```

Voir aussi ce bug, lié à **Get-Command**, lorsqu'il existe deux cmdlets de même nom :
https://connect.microsoft.com/PowerShell/feedback/ViewFeedback.aspx?FeedbackID=482523

9.5 Une approche du scripting basée sur des plug-ins

Sur le blog de MS-PowerShell, James Brundage propose une approche de développement de script basée sur un mécanisme de plug-in :

http://blogs.msdn.com/powershell/archive/2008/12/25/get-commandplugin.aspx

La fonction avancée *Get-CommandPlugin* utilise les métadonnées pour exécuter un ensemble de commandes basées sur une signature commune :

```
 #Affiche les plug-ins Get-CodeFromxxx
Get-CommandPlugin (gcm  Get-Code) -preposition "From"
 #La fonction Get-Code exécute les plug-ins précédents
Get-code http://blogs.msdn.com
```

10 Proxy de cmdlet

Nous avons rapidement abordé le principe des métadonnées sous PowerShell version 2, elles décrivent des cmdlets, des scripts ou des fonctions, on peut donc connaître les détails internes d'une commande. Sous PowerShell une métadonnée n'est pas inerte, elle n'est pas qu'une description, on peut interagir avec elles pour écrire des programmes qui manipulent des programmes, appelé aussi la méta programmation.

Une autre des avancées de cette version concerne l'écriture de proxy de cmdlet, le principe est d'encapsuler une commande, compilée ou pas, afin de modifier son comportement. Par exemple, restreindre les fonctionnalités d'un cmdlet en supprimant un ou plusieurs de ces paramètres, ou ajouter une fonctionnalité.

Les métadonnées d'une commande sont hébergées, au travers des classes dérivées de ***CommandInfo***, dans un champ d'accès privé de type ***CommandMetadata***. Il est possible de créer une instance de ce type :

```
$MetaData= New-Object System.Management.Automation.CommandMetaData(
              Get-Command Get-Item )
```

On utilise son constructeur suivant :

```
public CommandMetadata ( CommandInfo commandInfo )
```

L'affichage de notre variable *$MetaData* nous renvoie quelques informations supplémentaires, notamment la valeur de l'attribut *ConfirmImpact*, la plupart étant déjà présentes dans les classes dérivées de ***CommandInfo***.

L'attrait de cette instance est qu'elle peut être utilisée pour créer une commande basée sur une autre commande. On la passe en argument au constructeur statique de la classe ***ProxyCommand***:

```
$MetaProg=[System.Management.Automation.ProxyCommand]::create($MetaData)
```

Vous pouvez aussi utiliser une redirection dans un fichier pour récupérer le code généré :

```
$MetaProg > ProxyGetItem.ps1
```

L'affichage de la variable *$MetaProg* nous indique que cet objet contient du code :

```
$MetaProg
```

```
[CmdletBinding(DefaultParameterSetName='Path', SupportsTransactions=$true)]
param(
    [Parameter(ParameterSetName='Path', Mandatory=$true, Position=0, ValueFromPipeline=$true,
ValueFromPipelineByProper
tyName=$true)]
    [System.String[]]
    ${Path}, …
```

Ce code correspond, au moins pour les paramètres, à la transcription du code C# du cmdlet **Get-Item :**

Disassembler

```
[Cmdlet("Get", "Item", DefaultParameterSetName="Path")]
public class GetItemCommand : CoreCommandWithCredentialsBase
{
  // Properties
  [Parameter(Position=0, ParameterSetName="Path", Mandatory=true, ValueFromPipeline=true, ValueFromPipelineByPropertyName=true)]
  public string[] Path { get; set; }
```

Cette transcription, générée en interne à l'aide des méthodes *GetBegin*, *GetParamBlock*, *GetCmdletBindingAttribute*, *GetHelpComments*,…, utilise pour les noms de variable la notation **${Path}**, elle est identique à **$Path**, la différence étant, pour la première, le possible usage d'espaces dans le nom de la variable.

La variable *$MetaProg* est bien un proxy (ou stub). Son principe est de mettre à disposition du développeur "l'enveloppe" d'une commande, ce qui lui permettra de la modifier selon ses besoins. Le proxy de commande ne permet pas de modifier le code de la commande qu'il enveloppe, mais juste son paramétrage, il ne s'agit donc pas d'héritage de cmdlet. Le proxy de commande est un intermédiaire à qui l'on confie une charge, mais c'est à vous de définir cette charge.

Le code de ce proxy est constitué d'une signature, la liste des paramètres, et des blocs *Begin*, *Process* et *End*.

PowerShell version 2 propose un mécanisme d'accès au pipeline de la commande concernée, le terme US utilisé est *steppable pipeline* que je traduirais par pipeline 'pas à pas' ou pipeline séquencé.

10.1 Steppable Pipelines

Un pipeline 'pas à pas' relie le pipeline de la commande encapsulée au pipeline du code de votre proxy et vous autorise à contrôler les blocs *Begin*, *Process* et *End* de la commande concernée.

On doit impérativement utiliser un Scriptblock pour créer ce type de pipeline.

Reprenons le code généré précédemment :

```
$MetaProg > ProxyGetItem.ps1
```

10.1.1 Bloc Begin

Ce bloc prépare les paramètres additionnels et la gestion du pipeline. Occupons-nous dans un premier temps de celle-ci.

Les premières lignes gèrent le paramètre *OutBuffer*, puis on récupère les métadonnées de la commande concernée en précisant son nom et son type (***System.Management.Automation.CommandTypes***). :

```
$wrappedCmd = $ExecutionContext.InvokeCommand.GetCommand('Get-Item','Cmdlet')
```

La suivante crée un Scriptblock, son contenu exécutera la commande en lui passant les paramètres liés :

```
$scriptCmd = {& $wrappedCmd @PSBoundParameters }
```

C'est donc notre fonction proxy qui déclenche l'exécution de la commande, ici ***Get-Item***.

Le Scriptblock ne doit contenir qu'une commande ou un pipeline :

```
$scriptCmd = {"Test sans pipeline, ni commande"}
```

Sinon PowerShell déclenchera l'exception suivante lors de l'exécution du proxy :

> Exception lors de l'appel de « GetSteppablePipeline » avec « 1 » argument(s) : « Ne peut convertir qu'un bloc de script contenant exactement un pipeline ou une commande. Les expressions ou les structures de contrôle ne sont pas autorisées. Vérifiez que le bloc de script contient exactement un pipeline ou une commande.

L'avant-dernière ligne obtient, du Scriptblock *$ScriptCmd*, l'accès au mécanisme du pipeline de la commande :

```
$steppablePipeline = $scriptCmd.GetSteppablePipeline($myInvocation.CommandOrigin)
```

Notez que ce Scriptblock n'a pas été exécuté. On lui passe en argument l'origine de notre commande (*Runspace*).

Et enfin la dernière ligne appel le bloc *Begin* de notre commande en lui passant en argument la variable automatique *$PSCmdlet* :

```
$steppablePipeline.Begin($PSCmdlet)
```

Celle-ci sert à configurer notre commande avec les informations de contexte d'exécution de notre fonction. C'est donc cette ligne qui exécute notre commande en 'pas à pas'.

Comme tout bloc ***Begin***, ici aussi il est exécuté une seule fois, dans ce cas on exécute le début du bloc ***Begin*** du proxy, puis l'intégralité de celui de la commande et enfin on revient terminer celui du proxy. Ainsi, on lie bien le bloc ***Begin*** du proxy à celui de la commande encapsulée.

L'enchaînement de ce bloc au sein d'un pipeline ne diffère pas de la version 1 de PowerShell.

10.1.2 Bloc Process

Le bloc process est quant à lui réduit à sa plus simple expression. On exécute le bloc ***Process*** de notre commande au travers du pipeline 'pas à pas', *$steppablePipeline.* Cette méthode propose plusieurs surcharges, ici on lui passe en argument l'objet que notre fonction proxy est susceptible de recevoir du pipeline, si toutefois notre commande en attend un :

```
process
{
    try {
        $steppablePipeline.Process($_)
    } catch {
        throw
    }
}
```

Le bloc ***try-catch*** gère toutes les exceptions pouvant être déclenchées.

Il faut noter au moins trois choses sur le comportement de ce 'type' de pipeline :

1. les instructions situées après l'appel *de $steppablePipeline.Process* sont exécutées une fois que le bloc ***Process*** de la commande a émis dans le pipeline, non pas un, mais tous les objets qu'il doit traiter. Et ceci, pour chaque objet reçu par le pipeline du proxy. C'est le comportement normal du pipeline.
2. Le résultat de *$steppablePipeline.Process* est automatiquement émis dans le pipeline construit lors de l'appel du proxy et pas dans un pipeline indépendant, ce qui fait qu'on ne peut pas le récupérer. On ne peut manipuler que les objets émis en amont du pipeline, ceux émis par le pipeline 'pas à pas' ne nous sont pas accessibles, mais le sont pour les segments suivants. Le 'pas-à-pas' ne se fait pas sur les objets renvoyés par la commande, mais uniquement sur ceux reçus par le proxy.
3. Si vous appelez plusieurs fois la méthode *Process* du pipeline pas à pas, l'objet ou les objets seront émis plusieurs fois.

L'enchaînement de ce bloc au sein d'un pipeline ne diffère pas de la version 1, en revanche l'usage de la méthode **Process,** de la classe ***SteppablePipeline***, n'itère pas sur les éléments que la commande encapsulée renvoi.

10.1.3 Bloc End

Ce bloc exécute le bloc ***End*** de la commande, comme tout bloc ***End*** il est exécuté une seule fois lors de la finalisation du pipeline :

```
end
{
    try {
        $steppablePipeline.End()
    } catch {
        throw
    }
}
```

L'enchaînement de ce bloc au sein d'un pipeline ne diffère pas de la version 1. Par contre, toutes les instructions situées après l'appel de *$steppablePipeline.Process* de chaque proxy seront exécutées les une à la suite des autres avant ce bloc.

Et enfin, il reste le bloc d'aide, contenant simplement des instructions de redirection vers l'aide d'origine de la commande utilisé dans le proxy :

```
<#

.ForwardHelpTargetName Get-ChildItem
.ForwardHelpCategory Cmdlet

#>
```

Note :

Si vous encapsulez une commande utilisant uniquement le bloc *End*, tel que **Sort-Object**, alors les 3 trois blocs sont exécutés, le bloc *Process* reçoit bien tous les objets du pipeline, mais l'appel à *$steppablePipeline.Process($_)* n'a aucun effet. Dans ce cas l'enchaînement de ces blocs au sein d'un pipeline ne diffère pas de la version 1.

10.2 Gestion des erreurs

Les erreurs non-bloquantes n'influent pas sur le fonctionnement d'un proxy de commande.

Une exception est déclenchée si la commande encapsulée n'existe pas ou plus.

```
TestProxyFunctionPause
```

```
Le terme « TestProxyFunctionPause » n'est pas reconnu comme nom d'applet de commande, fonction, fichier de script ou programme exécutable. Vérifiez l'orthographe du nom, ou si un chemin d'accès existe, vérifiez que le chemin d'accès est correct et réessayez.
```

Une exception quel qu'elle soit provoquera l'arrêt du pipeline.

Si, par exemple, vous placez une instruction ***Break*** dans le bloc *Process*, alors l'enchaînement du pipeline est rompu. Dans ce cas les blocs *End* ne seront pas exécutés, mais aucune erreur n'est générée.

10.3 Suppression de paramètres

Il suffit de ne pas proposer dans la clause ***Param*** du proxy, le ou les paramètres concernés. Ainsi, on est assuré que lors de la liaison le ou les paramètres ne pourront être transmis.

10.4 Ajout de paramètres

L'ajout de paramètre vous oblige à coder la logique associée et à vous assurer que le ou les nouveaux paramètres ne seront pas transmis à la commande encapsulée, car cela provoquerait une erreur lors de la liaison. Prenons comme exemple celui proposé par Jeffrey Snover dans ce post :

Extending and/or Modifing Commands with Proxies

http://blogs.msdn.com/powershell/archive/2009/01/04/extending-and-or-modifing-commands-with-proxies.aspx

Pour gérer le paramètre additionnel *$SortBy*, il procède de la manière suivante :

Il ajoute le paramètre additionnel dans la liste des paramètres :

```
param(
    ...
    ${SortBy}) #Peut contenir un objet ou un tableau d'objets
...
```

Puis, dans les premières lignes du bloc *Begin* du proxy, il teste la présence du paramètre additionnel, s'il existe il le supprime de la liste des paramètres et construit le Scriptblock du nouveau traitement :

```
if ($SortBy)
 {
    [Void]$PSBoundParameters.Remove("SortBy")
    $scriptCmd = {& $wrappedCmd @PSBoundParameters |
                    Sort-Object -Property $SortBy
                 }
 }
```

Notez que la variable *ScriptCmd* contient un pipeline utilisant la commande **Sort-Object** qui propage le nouvel argument du proxy *via* son argument *-Property*. Aucun code n'est ajouté à la commande d'origine, mais on spécialise tout de même le comportement de la commande encapsulée.

Si le paramètre additionnel n'est pas précisé, alors la commande est exécutée normalement :

```
else
{ $scriptCmd = {& $wrappedCmd @PSBoundParameters }  }
```

Vous trouverez également, dans le post cité, un module facilitant la création d'instance de type ***ParameterAttribute*** et de proxy de commande.

Vous pouvez utiliser un script ou une fonction pour héberger votre proxy, de mon coté je préfère utiliser une fonction dans un module, qui est aussi un fichier, ce qui simplifie, entre autres, le chargement et le déchargement du proxy.

Si vous nommez votre fonction avec le même nom que la commande d'origine, c'est la fonction qui sera prioritaire sur la commande d'origine.

D'autres exemples :

Invoke-Command Proxy :

http://nullreference.spaces.live.com/blog/cns!C6138C01B9E969DF!1187.entry

Add filepath to convertto-html :

http://dmitrysotnikov.wordpress.com/2009/10/08/add-filepath-to-convertto-html/

Proxy for import-module :

http://richardsiddaway.spaces.live.com/Blog/cns!43CFA46A74CF3E96!2703.entry

Un wrapper pour Out-Default, par Bruce Payette :

http://poshcode.org/803

Comment implémenter Dir /a:d ?

http://blogs.msdn.com/powershell/archive/2009/03/13/dir-a-d.aspx

Ce code d'exemple propose également une gestion des paramètres dynamiques de la commande encapsulée.

11 Scriptblock et attributs

Un Scriptblock peut utiliser des attributs et accède à la variable automatique *$PScmdlet* :

```
$sb={
  param (
    [Parameter(Position=0, Mandatory=$false,ValueFromPipeLine=$true)]
   [string] $Nom = "ScriptBlcok"
  )
  $pscmdlet|gm|Sort membertype,name
}

&$sb
```

Il est donc possible d'utiliser des "Scriptblock avancées" implémentant *ShouldProcess* ou des paramètres dynamiques, etc.

Notez que ses métadonnées, de type **ScriptInfo,** ne semblent pas accessibles nativement, mais Il reste possible de convertir ce Scriptblock en une fonction.

12 Construire l'aide en ligne

Une autre avancée concerne l'intégration de la documentation au sein du code d'une fonction ou d'un script, aussi appelé AutoHelp.

Deux nouveaux tokens délimitent une zone de documentation multi lignes : <# et #>

La structure de ce type d'aide intégrée est la suivante :

```
<#
.< mot clé d'aide>
< contenu d'aide associé>
#>
```

Il est possible de définir la documentation en utilisant des commentaires mono ligne :

```
# .< mot clé d'aide>
# <contenu d'aide associé >
```

La présence des caractères < >, en dehors d'un commentaire multi lignes, sert ici de délimiteur et ne fait pas partie la syntaxe :

```
<#
.Example
PS C:\> MaCommande $Variable|Select –First 4
Etc.
#>
```

Certains mots clés d'aide, tel que ***Example***, peuvent être précisés plusieurs fois, d'autres sont à usage unique.

Consultez le fichier 'about_Comment_Based_Help.txt' pour le détail de cette convention et les règles de formatage qui s'y appliquent.

L'usage de certains paramètres tels que *Role, Component* et *Fonctionality*, du cmdlet **Get-Help** ne fonctionne pas, en tout cas on ne sait pas trop comment les utiliser, à part de cette manière :

```
Get-Command |Foreach {(get-help $_.name).Component}
```

Donc, n'hésitez pas à consultez, le site MsConnect car cette fonctionnalité contient quelques bugs. De plus la prise en compte des retours à la ligne ne semble possible qu'avec un format de fichier ASCII.

12.1 Construire un fichier d'aide localisé au format MAML

Une autre possibilité facilitant la localisation est d'utiliser un fichier XML respectant le schéma MAML (*Multi-Agent Modeling Language*) et contenant les mêmes rubriques.

Chaque fichier d'aide localisé doit être placé dans un répertoire portant le même nom que celui de la culture ciblée (fr-Fr) ou une culture neutre (fr). Le fichier d'aide par défaut sera placé dans le répertoire du script, il sera utilisé s'il n'existe pas de fichier localisé pour la culture courante.

Nous aurons donc, pour le script Test.ps1, l'arborescence suivante :

```
C:\Scripts\Test.ps1
  #une seule culture ciblée
C:\Scripts\de-DE\Test-Help.xml    #Culture allemande
C:\Scripts\en-US\Test-Help.xml    #Culture américaine
  #plusieurs cultures ciblées
C:\Scripts\en\Test-Help.xml        #Cultures anglaises
C:\Scripts\fr\ Test-Help.xml       #Cultures françaises
C:\Scripts\Test-Help.xml          #Culture 'invariante'
```

La liste des noms de culture : http://msdn.microsoft.com/fr-fr/library/system.globalization.cultureinfo.aspx

Pour préciser le nom du fichier d'aide externe on utilise la balise de commentaire ***.ExternalHelp***

```
# .ExternalHelp C:\Scripts\Test-Help.xml
# .ExternalHelp C:\Scripts\Script.ps1.xml
```

Les contraintes sont les suivantes :

- le début du nom de fichier d'aide doit correspondre au nom complet du module, de la fonction ou du script avec son extension,
- le chemin doit être un nom de chemin complet,
- il ne peut pas référencer de variable,
- ni contenir d'espace (bug ?). Si c'est le cas, utilisez un nom de fichier DOS 8.3. Sous Cmd.exe utilisez la commande Dir /x.

  Vous pouvez aussi utiliser le script ..*Help**GetShortPath.ps1* proposé dans les sources. Il n'est pas nécessaire de préciser le nom complet au format 8.3, seuls les répertoires ou le nom de fichier comportant un espace peuvent l'être.

Par convention, le nom du fichier XML peut être postfixé avec **–Help**.

La valeur de l'argument de la balise ***ExternalHelp*** (*C:\MyScripts\MyHelpfile.xml*) semble résolue de la manière suivante :

- Le nom de fichier est séparé du reste du chemin. Ce chemin (C:\MyScripts) devient le dossier de départ pour la recherche.
- Le système d'aide recherche un dossier portant le nom de la culture d'interface du système d'exploitation (UI culture), telle que "en-US " ou "fr-FR ", dans les sous-dossiers du répertoire de démarrage. (C:\MyScripts\fr-FR).
- Si le fichier MAML spécifique n'est pas trouvé, alors la logique de recherche de langue de secours est appliquée et les recherches du système d'aide se font sur les dossiers portant un nom de langue alternative, tel que "en " ou "fr ". (C:\MyScripts\fr)
- Si le fichier MAML n'est toujours pas trouvé, le système d'aide le recherche alors dans le dossier de départ (C:\MyScripts\MyHelpFile.xml)

À noter que le chemin d'accès du fichier d'aide semble être mis en cache par PowerShell, il faut donc, lors de la mise au point de l'aide, recharger le script ou recréer l'objet pour la prise en compte des modifications du fichier XML.

Pour vous aider à construire le squelette d'un fichier XML MAML, vous pouvez utiliser les scripts suivants :

New-MAML : http://www.poshcode.com/1338

New-XML : http://www.poshcode.com/1244

Lors de mes tests de localisation, j'ai constaté que la modification de la culture du thread courant n'influençait pas la recherche du fichier d'aide localisé. L'usage de la balise *ExternalHelp* dans un script ne fonctionne pas correctement :

```
Get-Help "$pwd\Script.ps1"
```

Seul le fichier XML présent dans le même répertoire est pris en compte, de plus la présence d'un répertoire ayant pour nom celui de la culture courante empêchera le chargement de ce fichier d'aide.

Voir aussi :

How to Write Cmdlet Help : http://msdn.microsoft.com/en-us/library/aa965353(VS.85).aspx

Creating the Cmdlet Help File : http://msdn.microsoft.com/en-us/library/bb525433(VS.85).aspx

*User Interface Language Management (**Language Fallback in the Resource Loader**) :*

http://msdn.microsoft.com/en-us/library/dd374098(VS.85).aspx

La page suivante contient de nombreux lien sur le sujet :

http://www.codeplex.com/wikipage?ProjectName=DocProject&title=Sandcastle%20Help

Utilisation de la propriété InvariantCulture : http://msdn.microsoft.com/fr-fr/library/4c5zdc6a(VS.80).aspx

Exemple de fichier d'aide d'un module Powershell V2 :

http://stackoverflow.com/questions/1432717/powershell-v2-external-maml-help

Il est également possible d'utiliser une page web comme une aide online, consultez les posts suivants :

http://www.leeholmes.com/blog/GetHelpNdashOnline.aspx

http://chadwickmiller.spaces.live.com/Blog/cns!EA42395138308430!846.entry

13 Liens

Recommandations de développement de l'équipe Microsoft PowerShell :

http://blogs.msdn.com/powershell/archive/2009/04/16/increasing-visibility-of-cmdlet-design-guidelines.aspx

Cmdlet Development Guidelines : http://msdn.microsoft.com/en-us/library/ms714657(VS.85).aspx

14 Conclusion

Il reste sûrement des cas à étudier, ce tutoriel vous donne de nombreux éléments pour les réaliser de votre côté.

Ces nouvelles possibilités tirent le scripting vers le "haut" en simplifiant le développement de traitements spécialisés. L'équipe de développement de PowerShell nous démontre qu'il est possible de faire simple, on se concentre sur le quoi et pas sur le comment. Toutefois, ces nouvelles possibilités nécessitent un temps d'apprentissage conséquent, mais le bénéfice en vaut la chandelle et vous ne coderez plus de la même manière.

J'ai pu constater à maintes reprises que la documentation offline du produit est incomplète, voire erronée. Pour aborder ces nouveautés l'aide fournie ne contient pas les détails importants quant au fonctionnement, ni d'exemples conséquents. Je me suis souvent appuyé sur Reflector pour trouver les informations manquantes, sans lui il m'aurait été difficile d'approfondir certains points.

La compréhension du fonctionnement ne pose pas de problème particulier, à mon avis la difficulté d'un tel codage réside dans la connaissance, ou plutôt dans la mémorisation des différentes règles qui peuvent s'additionner.

Ce type de développement peut vous amener à aborder le SDK et les comportements sous-jacents qui sont d'habitude transparents ou peu exploités.

En même temps rien ne vous oblige à implémenter toutes ces nouveautés dans vos prochains scripts. De savoir qu'elles existent et quels sont leurs usages, vous permettra de les aborder progressivement.

Certains bugs cités à la fin du chapitre «Différences entre la version 1 et la version 2 », limite à mon avis l'usage des fonctions avancées. Par contre, je ne sais pas si Microsoft envisage de livrer prochainement un patch pour corriger ces quelques bugs mineurs.

Enfin en cas de problème, n'hésitez pas à consulter MSConnect ou la documentation online US qui est mise à jour suite aux remarques remontées sur MSConnect :

http://technet.microsoft.com/en-us/library/bb978525.aspx

Il reste, lié au code, d'autres nouveautés importantes telles que les modules, les événements, …

En attendant, bon dev !