Powershell : null ce n'est pas rien !

Par Laurent Dardenne, le 17/06/2015.

Niveau

Conçu avec Powershell v4 & v2 sous Windows Seven 64 bits.

Je tiens à remercier 6ratgus pour ses relectures technique.

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

Chapitres

1 A propos de Null

Sur Wikipédia (*Null et la logique ternaire***)** il est indiqué :

*« Le mot-clé **Null** fut introduit dans SQL pour exprimer les informations manquantes dans le modèle relationnel. L'introduction de Null, avec True et False est le fondement de la logique ternaire.*

Null n'a pas de valeur (et n'est membre d'aucun type de donnée), c'est un mot-clé réservé, indiquant qu'une information est manquante. Par conséquent, la comparaison avec Null, même avec Null lui-même, ne peut ni être True ni être False, elle est obligatoirement inconnue (Unknown)…»

Sous Powershell, à la différence du SQL, l'égalité sur **$null**, comme en C#, renvoie $True :

```
$null -eq $null
True
```

$null est une valeur fictif de type scalaire. Les spécifications de Powershell version 3 indiquent :

> *« 4.1.2 The null type*
>
> *The* **null type** *has one instance, the automatic variable $nu11 (§2.3.2.2), also known as the null value. This value provides a means for expressing "nothingness" in reference contexts. The characteristics of this type are unspecified. »*

Il s'agit donc d'une variable automatique en lecture seule.

Rappel :

- Un type référence nécessite un appel explicite à un constructeur et est alloué sur le tas (heap), par exemple un objet *Process* :

  ```
  $P=get-process -Name Powershell
  $P.GetType().IsValueType
  False
  ```

  Dans ce cas la variable $P référence l'adresse mémoire d'un objet *Process*.

- Un type valeur ne nécessite pas d'appel préalable à un constructeur et est alloué sur la pile (stack), par exemple un *Integer*.

  ```
  $i=10
  $i.GetType().IsValueType
  True
  ```

Voir aussi : Types valeur et référence.

Un type valeur ne peut contenir la valeur **$null** :

```
[Int]$I=$null
$I
0
```

Ici le framework dot Net affecte la valeur par défaut du type [Int] à la variable $I, c'est-à-dire zéro. Notez que cette valeur n'est pas égale à **$null** :

```
$I -eq $null
False
```

1.1 Note sur l'opérateur –eq

Pour un scalaire, par exemple un entier, l'opérateur **–eq** valide ou non l'égalité sur la valeur $null, si on l'utilise sur une collection la validation se fait sur tous ses éléments :

```
[Int[]]$T=1,2,3,1
$T -eq 1
1
1
$T -eq $null
#Aucun affichage
```

Si on adresse directement l'objet dotNet adapté l'opérateur renvoi la bonne information :

```
$T.psbase -eq $null
False
$T=$null
$T -eq $null
True
$T.psbase -eq $null
True
```

Si on utilise une variable intermédiaire mémorisant le résultat du test, elle contiendra un tableau :

```
$Result=$T -eq $null
$Result.PSTypeNames
System.Object[]…
$Result.Count
0
```

Note : La présence de trois points supprime un affichage jugé verbeux.

L'approche suivante fonctionne dans tous les cas :

```
[Int[]]$Tab=1,2,3,1
if ($Tab) { Write-Host '$Tab n''est pas null' }
$Tab n'est pas null
$Tab=$null
if (!$Tab) { Write-Host '$Tab est null' }
$Tab est null
```

Note : le point d'exclamation est un raccourci de l'opérateur logique **–not**.

Sous Powershell le type de l'opérande gauche détermine comment l'opération se déroule. Dans l'exemple suivant, utilisant un opérateur arithmétique, l'opérande de gauche est de type [int], l'opérande de droite est, si possible, converti en [int] :

```
10+'1'
11
```

Si on inverse les opérandes, l'opérande de droite est converti en type [string]

```
'10'+1
101
```

Dans le cas suivant l'opérateur d'égalité peut renvoyer une valeur différente selon l'écriture :

```
$a = 1
$b = '01'
$a -eq $b
True
$b -eq $a
False
```

Donc pour tester si un tableau est $null on doit inverser les opérandes :

```
Remove-Variable T -EA SilentlyContinue
$T=1,2,3,1,$null,$null
$T -eq $null
#Aucun affichage
$null -eq $T
False
```

$null étant de type scalaire, Powershell teste ici l'adresse du tableau et pas son contenu.

Enfin, bien que le test sur la valeur $null ne déclenche pas d'affichage sur la console, notez que l'opérateur –eq fonctionne correctement :

```
$Result=$T -eq $null
$Result.Count
2
```

Ceci est dû au fait que PS ne déclenche pas d'affichage pour la valeur $null.

1.2 Itération sur $null (PS v2)

Dans Powershell version 2 la construction suivante est exécutée une fois :

```
Foreach ($Value in $null) {"Hello"}
Hello
```

$null, bien qu'étant une valeur particulière, est émis dans le pipeline.

1.3 Affectation de $null dans une variable

Tester la valeur $null sur une variable inexistante renvoi toujours $true :

```
Remove-Variable NotExist -EA SilentlyContinue
Test-Path Variable:NotExist
False
$NotExist -eq $null
True
```

Si on renforce le contrôle des règles d'analyse du code, le comportement change :

```
Set-StrictMode -Version latest
PS>$NotExist -eq $null
Impossible d'extraire la variable « $NotExist », car elle n'a pas été définie.
Set-StrictMode -Off
```

L'affectation de la valeur $null crée la variable si elle n'existe pas :

```
$NotExist=$null
Test-Path Variable:NotExist
True
```

Cette initialisation peut quelque fois être nécessaire afin de ne pas dépendre du mode d'analyse stricte. Il est inutile d'affecter $null à une variable en fin d'exécution d'un script ou d'une fonction, le fait de quitter la portée courante supprimera les variables créées. En revanche on doit forcer l'appel à la méthode *Dispose()* ou *Close()* pour les objets Disposable.

1.4 Affectation d'un résultat dans $null

L'instruction suivante, bien que la variable $null soit en lecture seule, ne génère pas d'erreur et ne modifie pas le contenu de la variable $null :

```
$null=Get-Process
$null
#Aucune valeur n'est émise dans le pipeline
```

Cette construction est identique à celles-ci

```
Get-Process > $null
Get-Process |Out-Null
[void](Get-Process)
```

Ceci permet de ne pas tenir compte du résultat de l'instruction. Seul le temps d'exécution est impactée, sachez que le cmdlet **Out-Null** est le plus lent.

1.5 A propos de Void

Le texte de la définition du 'type null', issu des spécifications de Powershell, utilise le terme *no-thingness*, dans ce contexte on peut le comprendre comme absence de valeur. Une variable étant une association (Nom, Valeur), lui affecter $**null** défini une variable sans valeur.

Powershell, comme le C#, ne distingue pas une méthode d'une fonction :

- Méthode : exécute du code et peux retourner un résultat,
- Fonction : exécute du code et retourne toujours un résultat.

Pour les cas où on ne souhaite pas tenir compte du résultat d'une fonction, on utilisera le raccourci [Void] qui précise une absence de résultat :

```
$List= new-object System.Collections.ArrayList
$List.Add(10)
0
$List
10
[void]$List.Add(5)
$List
10
5
$List.Add(7)
2
```

Pour le second appel à la méthode *Add()*, l'usage de **[void]** ne tient pas compte de la valeur de retour, elle n'est pas émise dans le pipeline.

Pour savoir si une méthode émet ou non des données, on doit afficher ses signatures :

```
$List.Add
OverloadDefinitions
-------------------
int Add(System.Object value) …
```

Ici **int** indique une valeur de retour de type entier. Il s'agit de l'index de la nouvelle valeur dans la collection $List.

On peut obtenir le détail la méthode *Add* de la manière suivante :

```
$List.GetType().GetMember('Add')
Name                        : Add
MemberType                  : Method
ReturnType                  : System.Int32
Attributes                  : PrivateScope, Public, Virtual, HideBySig, …
```

Dans l'exemple suivant la méthode ne renvoie aucune valeur :

```
'ChaîneDeCaractères'.CopyTo
OverloadDefinitions
-------------------
void CopyTo(int sourceIndex, char[] destination, int destinationIndex,…
```

Void précise que la méthode *CopyTo* ne renvoie aucun résultat. Ici l'usage du raccourci [***Void]*** serait redondant. L'appel d'une telle méthode n'émet pas d'objet dans le pipeline.

<u>[Void] est une directive destinée au parseur (PS) ou au compilateur (C#)</u>.

1.6 Pourquoi ne pas utiliser [void] comme valeur de retour ?

On peut se dire : '*puisse que $null est une valeur renvoyons* ***[Void]*** *afin d'indiquer que nous ne renvoyons pas de valeur*' :

```
Foreach($Value in [void]) {Write-Warning $value.gettype() ; "Hello"}
AVERTISSEMENT : System.RuntimeType, mscorlib, Version=4.0.0.0, Cul-
ture=neutral, PublicKeyToken=b77a5c561934e089
Hello
```

Bien que [void] ne soit pas égale à $null :

```
 #exécution d'un scriptblock
$V=&{ Return [void] }
$V -eq $null
False
```

Cela ne fonctionne pas car on émet un objet représentant le type de la classe [void]. Sachez que ce raccourci ne peut être utilisé pour typer une variable :

```
[void]$V=10
[void] ne peut pas être utilisé en tant que type de paramètre ou à gauche
d'une affectation.
```

L'erreur déclenchée par le code suivant est tout autant explicite :

```
[void]10| Foreach {write-output 'Ok'}
Une erreur s'est produite lors de la création du pipeline.
Resolve-Error
...
 ---> System.ArgumentException: Le type de l'argument ne peut pas être
System.Void.
```

1.7 Fonction ne renvoyant pas de résultat (PS v2)

Avec PowerShell version 2, si on affecte à une variable le résultat d'une fonction ne retournant pas de valeur, elle contiendra la valeur *$null* :

```
Function RienAEmettre {}
$o=RienAEmettre
$O -eq $null
True
Foreach($Value in $O) {"Hello"}
Hello
RienAEmettre|Foreach-Object {'hello'}
Hello
```

Mais si on utilise directement la fonction, celle-ci n'émet aucun objet dans le pipeline :

```
RienAEmettre|Foreach-Object {'hello'}
Foreach($Value in RienAEmettre) {"Hello"}
#Aucun affichage
```

1.7.1 Déclenchement d'exception lors d'une affectation de variable

Lors de son exécution une fonction peut déclencher une exception :

```
function Exception {
 param([Switch]$ThrowException)
  if ($ThrowException)
  {throw "Erreur"}
}
```

Dans le cas où la variable ciblée par l'affectation n'existe pas Powershell ne la crée pas :

```
Remove-Variable V -EA silentlycontinue
$v=Exception -ThrowException
Erreur
Write-host "isNull : $($v -eq $null)"
isNull : True
Test-Path variable:V
False
```

Si la variable ciblée existe, Powershell ne modifie pas son contenu :

```
Remove-Variable V -EA silentlycontinue ; $V='Before'
$v=Exception -ThrowException
Erreur
Write-host "isNull : $($v -eq $null)"
isNull : False
$V
Before
```

Si le traitement réussi et que la fonction appelée ne renvoie pas de résultat, toutes les versions de Powershell affectent la valeur $null à la variable :

```
$v=Exception
Write-host "isNull : $($v -eq $null)"
isNull : True
```

2 Faites un break !

A partir de Powershell version 3, si on reprend un des exemples précédent, l'itération ne se fait plus :

```
Function RienAEmettre {}
$o=RienAEmettre
$O -eq $null
True
Foreach($Value in $O) {"Hello"}
#Aucune itération
```

Ceci est documenté (ForEach statement does not iterate over $null) comme un breaking change, C'est-à-dire une modification du code interne qui, par propagation, a des répercussions sur le comportement des scripts existants, créés et validé avec une version précédente.

Voyons maintenant le comportement d'une fonction ne renvoyant pas de résultat :

```
function Test() {
    function getNull {$null}
    $v=getNull
    return $V
}
$Result = Test
Foreach($Value in $Result) {"Hello"}
#Aucune itération
$Result|Foreach-Object {"Hello"}
Hello
```

La variable $Result contient $null, si on utilise le cmdlet Foreach-Object au lieu de l'instruction Foreach on constate que sa valeur est émise dans le pipeline.

Autre exemple, on s'assure que la variable utilisée dans la fonction *Test* n'existe pas dans la portée courante, sinon sa valeur serait renvoyée :

```
RV NotExist -EA SilentlyContinue
function Test() {  return $NotExist}
$Result = Test
$Result|Foreach {"Hello"}
Hello
```

Bien que la variable $NotExist n'existe pas, $Result contient $null.

Dans l'exemple suivant, la fonction utilisé ne renvoie pas de résultat, on s'attend, comme avec la version 2 de Powershell, à recevoir la valeur $null :

```
RV V -EA SilentlyContinue
function Test() {
    function GetNothing {}
    $V=GetNothing
    return $V
}
$Result = Test
$Result -eq $null
True
$Result|Foreach {"Hello"}
#Aucune valeur n'est émise dans le pipeline
```

D'après le test sur $null la variable $Result devrait déclencher une itération, il n'en est rien.

Désormais, afin d'indiquer à l'utilisateur qu'une fonction ne renvoie aucun résultat, powershell émet la valeur *[System.Management.Automation.Internal.AutomationNull]::Value*

Il s'agit donc d'un breaking change qui n'est pas détaillé.

Cette valeur est égale à la variable **$null,** mais elle n'est pas identique, étrange affirmation me direz-vous :

```
$null -eq [System.Management.Automation.Internal.AutomationNull]::Value
True
```

La variable **$null** bien qu'étant un scalaire n'est pas du type PSObject :

```
$null -is [PSObject]
False
```

Par contre, la valeur du singleton ***AutomationNull*** est du type PSObject:

```
[System.Management.Automation.Internal.AutomationNull]::Value -is [PSOb-
ject]
True
```

On ne peut pas pour autant récupérer son type :

```
$AutomationNull = [System.Management.Automation.Internal.AutomationNull]::Value
$AutomationNull.GetType()
Impossible d'appeler une méthode dans une expression Null.
```

Ce comportement reste cohérent avec celui de la variable $null. **A partir de la version 3**, on peut utiliser la propriété PSBase :

```
$AutomationNull.psbase.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     PSObject                                 System.Object
```

L'émission de la valeur $null dans le pipeline fonctionne toujours :

```
$null|Foreach {'suite du traitement'}
suite du traitement
```

Mais avec la version 3, dorénavant le pipeline filtre les variables dont le contenu est égal à *AutomationNull* :

```
$AutomationNull|Foreach {'suite du traitement'}
#Aucune valeur n'est émise dans le pipeline
```

Ainsi Powershell sait que pour une fonction ne renvoyant pas de résultat, il ne doit pas émettre la valeur *AutomationNull* dans le pipeline :

```
Function RienAEmettre {}
RienAEmettre |Foreach {'suite du traitement'}
#Aucune valeur n'est émise dans le pipeline
```

Le comportement est identique pour les versions 2 et supérieures.

En revanche celui-ci fonctionne avec la version 2, mais pas avec les versions supérieures :

```
$Result=RienAEmettre
$Result|Foreach {'suite du traitement'}
#V3 et > : Aucune valeur n'est émise dans le pipeline
#V2      : suite du traitement
```

Il s'agit d'un nouveau comportement, mais pas d'une nouvelle variable car ***AutomationNull*** est accessible dans la version 2, mais lors d'une affectation elle est convertie en interne en une valeur $null.

A la différence du SQL où *null* précise une information manquante, sous Powershell $null précise explicitement une absence de valeur.

Le singleton AutomationNull précise explicitement une absence de valeur et implicitement une absence de résultat.

2.1 Comportements de l'instruction Foreach

2.1.1 Itération sur $null (PS v3)

A partir de Powershell version 3 les constructions suivantes ne déclenchent plus d'itération :

```
$N=$null
$AN=[System.Management.Automation.Internal.AutomationNull]::Value
Foreach($i in $null) {"Hello"}
Foreach($i in $N) {"Hello"}
Foreach($i in $AN) {"Hello"}
```

2.1.2 Itération sur un tableau d'élément $null

Dans ce cas l'itération fonctionne :

```
Foreach($i in $null,$null) {"Hello"}
Hello
Hello
```

Les constructions suivantes ont un comportement identique :

```
Foreach($i in $N,$N) {"Hello"}
Foreach($i in $AN,$AN) {"Hello"}
#ou
$T=$AN,$AN
Foreach($i in $T) {"Hello"}
```

Elles renvoient toutes le même résultat :

```
Hello
Hello
```

Tout comme celle-ci :

```
Foreach($i in @($null,$null)) {"Hello"}
```

Sauf celle-là :

```
Foreach($i in @($AN,$AN)) {"Hello"}
#Aucune valeur n'est émise dans le pipeline
```

Avec la version 2, Powershell émet les deux objets.

La différence est que la syntaxe suivante appelle un constructeur pour créer le tableau :

```
Remove-Variable T -EA Silentlycontinue
$T=$AN,$AN
$T.Count
2
```

Alors que celle-ci récupère d'abord le résultat *via* le pipeline, puis crée un tableau

```
Remove-Variable T -EA Silentlycontinue
$T=@($AN,$AN)
$T.Count
0
```

Peu importe la syntaxe de construction du tableau, le pipeline filtre la valeur *AutomationNull* :

```
$T=$AN,$AN
$T|Foreach {'suite du traitement'}
#Aucune valeur n'est émise dans le pipeline
```

2.2 Void force l'émission de $null

L'usage de **[Void]** affecte toujours la valeur *$null* et pas la valeur *AutomationNull* :

```
$V=[void](&{10})
$V
#Aucun affichage
$V -eq $null
#True
$V -is [PSObject]
#False
$V|Foreach {'suite du traitement'}
#suite du traitement
```

C'est le même comportement si on l'utilise avec une fonction ne renvoyant pas de résultat :

```
Function RienAEmettre {}
$V=[void](RienAEmettre)
$V
#Aucun affichage
$V -eq $null
#True
$V -is [PSObject]
#False
$V|Foreach {'suite du traitement'}
#suite du traitement
```

Mais le transtypage d'une variable sans type en [void] affecte *AutomationNull* :

```
Remove-Variable V -EA Silentlycontinue
$v=10 -as [void]
$v -is [psobject]
True
```

Pour les autres cas, un échec de transtypage affectera *$null*. Ceci dit, il est inutile d'utiliser l'opérateur–**as** sur le type [void] pour obtenir une variable sans valeur.

2.3 Clear-Variable

Pour information, **Clear-Variable** affecte la valeur *$null :*

```
 #instructions exécutées à la suite du code précédent
Clear-Variable v
$v -is [psobject]
False
```

2.4 A propos de la variable automatique $input

Le type de la variable automatique *$input* diffère selon la construction utilisée :

```
function Test1 {
  param( $InputObject  )
  Write-Host "Test1 : $($input.gettype().Fullname)"
}

function Test2 {
  param(
    [Parameter(ValueFromPipeline=$True)]  $InputObject
  )
  Write-Host "Test2 : $($input.gettype().Fullname)"
}

function Test3 {
  param( $InputObject )
 process {
   Write-Host "Test3 : $($input.gettype().Fullname)"
 }
}

function Test4 {
  param(  [Parameter(ValueFromPipeline=$True)] $InputObject )
 process {
  Write-Host "Test4 : $($input.gettype().Fullname)"
 }
}

function Test5 {
 process {
  Write-Host "Test5 : $($input.gettype().Fullname)"
 }
}
```

```
Test1; Test2; Test3; Test4; Test5
Test1 : System.Collections.ArrayList+ArrayListEnumeratorSimple
Test2 : System.Object[]
Test3 : System.Collections.ArrayList+ArrayListEnumeratorSimple
Test4 : System.Collections.ArrayList
Test5 : System.Collections.ArrayList+ArrayListEnumeratorSimple
```

Selon la présence du bloc *process* et ou de la valeur d'attribut *ValueFromPipeline,* il peut être du type

- **System.Collections.ArrayList+ArrayListEnumeratorSimple,**
- **System.Object[]**,*
- ou **System.Collections.ArrayList ***.

* dans les spécifications de la version 3 (4.5.16) ces types ne sont pas mentionnés.

Avec la version 2 son type peut être :

- **System.Collections.ArrayList +ArrayListEnumeratorSimple**
- ou **System.Array+SZArrayEnumerator.**

L'itération sur cette variable via le pipeline ne change pas, comme vous pouvez le constater avec le code d'exemple de ce tutoriel. La migration de script PowerShell version 2 doit donc tenir compte du code utilisé autour de la variable $input.

Bien que le nombre d'élément de la collection $input varie (0 ou 1 selon les cas), cela n'influence pas l'itération, hormis en v2 si le code reçoit *AutomationNull* qui est transformé en *$null* :

```
function Test {
 param( [Parameter(ValueFromPipeline=$True)] $InputObject )
 process {
   $input|% { Write-host 'Itération $Input' -fore green }
   $InputObject|% { Write-host 'Itération $InputObject' -fore green }
 }
}
$O=Get-Process -Name NotExist -EA SilentlyContinue
$O|Test
#V3 et > : Aucune valeur n'est émise dans le pipeline
#V2      : Itération $Input
#V2      : Itération $InputObject
```

2.5 Valider la présence de AutomationNull

Jason Shirk a proposé sur StackOverflow la fonction suivante :

```
function Test-IsAutomationNull {
  param(
     [Parameter(ValueFromPipeline=$true)]
     $InputObject
  )
  begin  {
     if ($PSBoundParameters.ContainsKey('InputObject'))
     { throw "Test-IsAutomationNull only works with piped input"  }
     $isAutomationNull = $true
  }
  process {
   $isAutomationNull = $false
  }
  end {
   return $isAutomationNull
  }
}
$O=Get-Process -Name NotExist -EA SilentlyContinue
$O|Test-IsAutomationNull
True
$null|Test-IsAutomationNull
False
```

Son usage est limité au pipeline car l'émission d'*AutomationNull* ne déclenche pas le bloc **process,** alors que le bloc **begin** est toujours exécuté. L'émission de *$null* déclenchera le bloc **process.**

Par contre avec la syntaxe suivante *AutomationNull* est transformé en *$null* :

```
Test-IsAutomationNull -InputObject $O
```

Il est donc impossible de distinguer le type de la valeur d'origine.

Dans le post cité, l'auteur précise :

« *One might ask why $null is written to the pipeline. It's a reasonable question. There are some situations where scripts/cmdlets need to indicate "failed" without using exceptions - so "no result" must be different, $null is the obvious value to use for such situations.* »

3 Type Nulllable

Comme indiqué sur MSDN :

« Un type est dit Nullable s'il est possible de lui assigner une valeur ou de lui assigner null, ce qui signifie qu'il n'a aucune valeur du tout. Par défaut, tous les types référence, tels que String, sont Nullable, mais tous les types valeur, tels que Int32, ne le sont pas. »

Ce type implémente la logique ternaire sous dot Net, par exemple un booléen ne peut être que True ou False :

```
[Boolean]$O=$false
$null -eq $o
False
$O=$true
$null -eq $o
False
```

L'affectation de la valeur $null à un booléen déclenche une exception :

```
$O=$null
Impossible de convertir la valeur «» en type «System.Boolean». Les para-
mètres booléens acceptent seulement des valeurs booléennes et des nombres,
tels que $True, $False, 1 ou 0.
```

Déclarons un booléen nullable :

```
[System.Nullable[System.Boolean]]$O=$null
$null -eq $O
True
```

Ce type permet également la manipulation de données issues d'un SGBD qui elles peuvent être nulles.

La construction d'une variable de type nullable ne peut se faire que sur un type valeur :

```
[System.Nullable[System.Diagnostics.Process]]$O=$null
Type [System.Nullable[System.Diagnostics.Process]] introuvable. Assurez-
vous que l'assembly qui contient ce type est chargé.
Détails : GenericArguments[0], 'System.Diagnostics.Process', sur 'Sys-
tem.Nullable`1[T]' ne respecte pas la contrainte de type 'T'.
```

Le type process est un type référence il peut, de par sa conception, contenir $null.

Un entier 'classique' ne peut pas contenir $null :

```
[int32]$J=10
$J
10
$J=$null
0
```

Si on lui affecte *$null* Powershell le remplace avec la valeur par défaut du type, ici zéro.

Si on utilise un entier nullable la valeur *$null* est préservée :

```
[System.Nullable[System.Int32]] $I = 10
$I
10
$I.GetType().Fullname
System.Int32
$I=$null
$I
#Aucun affichage
```

Ici la variable $J est typé, on retrouve donc la conversion en une valeur par défaut :

```
[int]$J=2
$J=$I
$J
0
```

Si la variable n'est pas typée, celle-ci contiendra la valeur $null :

```
$K=2
$K=$I
$null -eq $K
True
```

Si on réaffecte à **$I** une valeur entière, son type ne change pas :

```
$I=5
$I.GetType().Fullname
System.Int32
```

Le type réel n'est pas affiché, à la place Powershell affiche le type d'origine transformé en un type nullable. Notez que l'affichage de l'aide d'une fonction possédant des paramètres de type nullable a le même comportement.

A la différence du C# on ne peut donc pas accéder à la propriété *HasValue* qui indique si l'objet Nullable actuel a une valeur valide de son type sous-jacent.

3.1 Variable contrainte sur un type

Dans le cas où l'on type une variable, Powershell le mémorise en ajoutant à sa liste d'attributs, l'attribut ***ArgumentTypeConverterAttribute*** :

```
$Var=Get-Variable I
$Var.Attributes
TypeId
------
System.Management.Automation.ArgumentTypeConverterAttribute
```

La fonction suivante permet de retrouver le type déclaré d'une variable :

```
function Get-VariableType {
# http://poshcode.org/998
  param([string]$Name)
    Get-Variable $name |
     Select-Object -expand attributes |
     Where {
        $_.GetType().Name -eq "ArgumentTypeConverterAttribute" } |
     Foreach {
        $_.GetType().InvokeMember("_convertTypes",`
         "NonPublic,Instance,GetField", $null, $_, @())
    }
}
$T=Get-VariableType I
$T
IsPublic IsSerial Name                      BaseType
-------- -------- ----                      --------
True     True     Nullable`1                System.ValueType
$A=$null
Get-VariableType A
#Aucun affichage, la variable $A n'est pas de typée
```

Cette fonction permet également de récupérer le type d'une variable contenant $null :

```
[PSobject]$A=$null
$A.GetType()
Impossible d'appeler une méthode dans une expression Null.
Get-VariableType A
IsPublic IsSerial Name                       BaseType
-------- -------- ----                       --------
True     True     PSObject                   System.Object
```

Pour connaitre le type sous-jacent d'un type nullable, on peut utiliser la méthode statique suivante :

```
[System.Nullable]::GetUnderlyingType($T)
IsPublic IsSerial Name                       BaseType
-------- -------- ----                       --------
True     True     Int32                      System.ValueType
```

Enfin pour tester si un type est d'un type nullable on peut utiliser cette fonction :

```
Function Test-NullableType {
 param([string]$Name)
  $T=Get-VariableType $Name
  If ($T)
  {
     Return ($T.IsGenericType -and ($T.GetGenericTypeDefinition() -eq
[Nullable`1]))
  }
  else
  { Return $False }
}
Test-NullableType A
False
Test-NullableType I
True
```

4 NullString

Dans Powershell la gestion d'un paramètre de type string, d'une fonction ou d'une méthode C#, converti $null en une chaîne vide. Ce qui fait qu'une méthode C# ne peut pas recevoir la valeur $null qui est différente d'une chaîne vide. On retrouve ce problème dans l'exemple suivant :

```
$content=@'
using System;
namespace ClassLibrary1
{
 public static class Class1
 {
  public static string Foo(string value)
  {
   if (value == null)
    return "Special processing of null.";
   else
    return "'" + value + "' has been processed.";
  }
 }
}
'@

Add-Type -TypeDefinition $content
```

Testons l'appel avec une chaîne vide et une chaîne qui ne l'est pas :

```
[ClassLibrary1.Class1]::Foo('NotEmpty')
'NotEmpty' has been processed.
[ClassLibrary1.Class1]::Foo('')
#ou [ClassLibrary1.Class1]::Foo([string]::Empty)
'' has been processed.
```

Les deux contenus affichent le résultat attendu, mais si on passe en paramètre la variable $null le code affichant '*Special processing of null'* n'est pas déclenché :

```
[ClassLibrary1.Class1]::Foo($null)
'' has been processed.
```

Afin d'éviter la conversion de $null en une chaîne vide, l'équipe Powershell a ajouté, dans la version 3 et supérieure, la valeur particulière :

[System.Management.Automation.Language.NullString]::Value

```
[ClassLibrary1.Class1]::Foo(
    [System.Management.Automation.Language.NullString]::Value)
Special processing of null.
```

Si on affecte cette valeur à une variable de type *String*, la valeur $null est conservée :

```
[string]$S=[System.Management.Automation.Language.NullString]::Value
$null -eq $s
True
```

Pour contourner ce problème sous Powershell version 2, on doit utiliser le système de réflexion pour construire l'appel :

```
$ParameterValues = @($null)
$FooMethod = [ClassLibrary1.Class1].GetMethod("Foo")
$FooMethod.Invoke([ClassLibrary1.Class1], $ParameterValues)
Special processing of null.
```

Dans notre cas, on passe en paramètre à la méthode *Invoke* :

1. le type déclarant la méthode statique, pour une instance on précise le nom de la variable,
2. un tableau de paramètres correspondant au nombre de paramètres de la méthode.

Il reste toutefois un problème avec la classe *NullString*, cette valeur n'est pas préservée lorsqu'elle est passée à une fonction, comme indiqué dans l'exemple suivant :

```
function Test {
 param ( [string] $Str )
 $null -eq $Str
 $Str -eq [String]::Empty
}
[string]$S=[System.Management.Automation.Language.NullString]::Value
Test $S
```

```
False
True
```