Les classes sous Powershell v5

Par Laurent Dardenne, le 06/11/2015.

Version 1.2

Niveau

Conçu avec Powershell version 5.0.10240.16384 - Windows 10 64 bits.

Je remercie Jason Shirk pour ses informations.

Je remercie également 6ratgus et Matthew Betton pour leur relecture technique.

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

Chapitres

1 Rappel

J'invite ceux et celles manipulant pour la première fois le concept de classe, à consulter le tutoriel 'La notion d'objet sous PowerShell'.

Ce document n'est pas un cours sur la programmation orienté objet (POO), il passe en revue certains de ces principes de base mise en œuvre sous Powershell version 5.

Comme on ne peut mouler de gaufre sans moule à gaufre, un objet ne peut être créé sans classe. Une classe est la définition d'une structure de données contenant des membres : champs, méthodes, propriétés.

Certains types sont définis par le système, par exemple le type entier, d'autres, des classes ou des énumérations, le sont par l'utilisateur ou par des produits tiers.

Jusqu'à maintenant il n'était pas possible de créer des classes en Powershell natif. Il existe bien le cmdlet **New-Object**, mais celui-ci crée un objet à partir d'un nom de classe existant quant au cmdlet **Add-Type** il compile du code C# ou VB ou F#.

A partir de la version 5, Powershell propose le mot clé ***Class*** permettant de définir une classe en utilisant la syntaxe de son langage :

```
Class NomDeClasse {
  #définition des membres de la classe
}
```

Note : Le copier-coller via le clic droit ne fonctionne pas correctement* dans la console avec le module PsReadline qui est désormais chargé avec le runtime. Une alternative est d'utiliser la combinaison de touches Ctrl+V.

* PsReadline exécute les lignes les unes après les autres et pas en un seul bloc.

Une fois la définition d'une classe parsée puis exécutée, le type de la classe est hébergé dans un assembly résidant en mémoire, et pas dans un fichier .dll.

La syntaxe d'accès à cette nouvelle classe est identique à toutes les autres :

```
[NomDeClasse]
[Int]
```

Comme à son habitude l'équipe de Powershell minimise l'apprentissage en réutilisant les connaissances acquises, la syntaxe est connu :

MotClé Identifiant { *Scriptblock* }

Pas de quoi casser trois pattes à un canard ;-)

2 A propos de type

Avant de détailler la construction d'une classe, regardons de plus près le type de notre classe Powershell :

```
$Properties=@('Name','Namespace','FullName')
[NomDeClasse] |
 Select-Object -Property $Properties |
 Format-List
```

```
Name                  : NomDeClasse
Namespace             :
FullName              : NomDeClasse
```

Comparons avec une classe dot Net :

```
[PSObject] |
 Select-Object -Property $Properties |
 Format-List
```

```
Name                  : PSObject
Namespace             : System.Management.Automation
FullName              : System.Management.Automation.PSObject
```

On constate que pour notre classe l'espace de nom n'est pas renseigné, pour le moment on ne peut pas déclarer d'espace de nom. Celui-ci regroupe et isole les classes, ce qui permet la réutilisation d'un nom de classe en évitant le problème de collision de nom.

Visualisons certains détails de la propriété *Assembly* :

```
[NomDeClasse].Assembly|select * -ExcludeProperty evidence
```

```
Location            :
FullName            : powershell, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null
DefinedTypes        : {NomDeClasse, <NomDeClasse_staticHelpers>}
ManifestModule      : powershell
IsDynamic           : True
Modules             : {<In Memory Module>}
…
```

Notre classe est hébergée dans un assembly dynamique, résidant donc en mémoire, qui déclare deux classes, la nôtre et une interne *<**NomDeClasse**_staticHelpers>* dont le nom reprend celui de notre classe. Seule la première nous intéresse.

On manipule bien une classe native dot Net, mais qui est en interne couplée au moteur d'exécution de Powershell. Ainsi on peut utiliser du code Powershell dans une déclaration de classe, par exemple pour mettre en œuvre (implémenter) le code d'une méthode.

Note : le module suivant permet d'enregistrer un assembly dynamique sur disque.

3 Introduction

Bien que la syntaxe provienne du C#, une déclaration de classe Powershell ne peut pour le moment définir que des propriétés et des méthodes. De plus **tous ses membres sont publics**.

Attention, les mots clé de Powershell ne peuvent être utilisés pour nommer un membre ou une classe.

Note : Sous dot Net, tous les champs, méthodes, constantes, propriétés et événements doivent être déclarés dans un type ; ils portent le nom de *membres* du type.

3.1 Constructeur

Chaque classe doit contenir un ou des constructeurs, leur rôle, schématiquement, est de créer et d'allouer l'espace mémoire nécessaire à un objet, aussi appelé ***instance***, puis d'initialiser ses propriétés. Pour cette opération on utilise également le verbe *instancier*.

Un constructeur est une méthode particulière dont le nom est identique à celui de la classe :

```
Class Test {
  Test() {
   Write-warning "Call constructeur par défaut"
  }
}
```

Il peut exister plusieurs constructeurs pour une même classe, chaque constructeur pouvant déclarer un nombre de paramètre différent :

```
Class Test {
  [string] $Property
  Test() {
   Write-warning "Call constructeur par défaut"
  }
  Test([string] $Parameter) {
   Write-warning "Call constructeur avec un paramètre: $Parameter "
   $this.Property=$Parameter
  }
}
```

La définition d'un constructeur ne contient pas de type retour puisque son rôle est de créer en mémoire une instance de sa classe, un objet. Implicitement cette méthode ne renvoie pas de résultat (voir le mot clé [Void]), son code ne peut donc contenir l'instruction *Return*.

Une fois la classe définie, il reste à appeler un constructeur pour créer un objet. On utilisera le raccourci d'écriture suivant :

```
$O=[Test]::new()
#ou
$O=[Test]::new('Nom')
```

Cette syntaxe appelle le constructeur et renvoi le nouvel objet. Ce raccourci d'écriture fonctionne avec toutes les classes :

```
$Date=[System.DateTime]::new(2020,1,1)
```

La syntaxe suivante affiche la liste des constructeurs publics :

```
[System.DateTime]::new
datetime new(long ticks)
datetime new(int year, int month, int day)
…
```

Une classe qui ne définit pas de constructeur se voit attribuer automatiquement un constructeur par défaut sans paramètre (*parameterless constructor*) :

```
Class Test { }
[Test]::new
OverloadDefinitions
-------------------
Test new()
[Test].GetConstructors() #Appel une méthode du type
Name      : .ctor
…
```

Le nom *.ctor* est une abréviation pour ***Contructor.***

Nous verrons que celui-ci peut s'avérer nécessaire dans certains cas. Une classe qui définit un constructeur ayant au moins un paramètre ne contiendra pas de constructeur par défaut, mais il reste possible de le déclarer intentionnellement.

Pour créer un objet on peut également utiliser le cmdlet **New-Object** :

```
$o=New-Object Test
```

Sachez que dans le code suivant :

```
[Test].GetConstructors()
```

[Test] est un objet particulier, il référence la définition d'une classe :

```
[Test]
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Test                                     System.Object
```

Vous pouvez considérer le terme *type* comme un synonyme du terme *classe*.

Ce code appel une méthode du type nommé **[Test]** afin d'afficher la liste de ses constructeurs.

3.2 Propriété

Elle contient une donnée associée à un objet :

```
Class Test {
   $Property
}
```

La déclaration du nom de propriété utilise la syntaxe d'un nom de variable, mais ne référence pas de variable :

```
$Property='Name'
Class Test { $Property }
```

Il n'y a pas non plus de substitution si une variable de même nom existe en dehors de la déclaration de la classe.

Affichons notre objet :

```
$o=[Test]::new()
$o
Property
--------
```

Notre propriété contient *$null*. Maintenant vérifions sa définition à l'aide de ***Get-Member*** :

```
$o|Get-Member -MemberType Property -Force
   TypeName: Test
Name     MemberType Definition
----     ---------- ----------
Property Property   System.Object Property {get;set;}
```

Ce membre est bien une propriété publique disposant des accesseurs **get** (lecture) et **set** (écriture). Son type est *System.Object*, celui par défaut si la propriété n'en précise pas.

3.2.1 Préciser un type

Il suffit, comme pour une variable, d'ajouter un nom de type devant le nom de la propriété :

```
Class Test {
 [string] $Property
}
[Test]::new()|Get-Member -MemberType Property -Force
…
Property Property   string Property {get;set;}
```

On ne peut pas utiliser le type ***[void]*** pour une propriété :

```
Class NomDeClasse {
 [void] $Property
}
The type 'void' is not allowed on a property.
```

3.2.2 Valeur par défaut

Une propriété peut définir une valeur par défaut, sinon elle reçoit celle du type :

```
Class Test {
 [string] $Property='Test'
 [Int]    $Nombre
 [System.IO.FileInfo] $File
}
$o=[Test]::new()
$o
Property Nombre
-------- ------
Test          0
```

La propriété nommée *Nombre* contient zéro, la valeur par défaut de la classe **[Int].** La propriété *File* contient ***$Null***, car on doit appeler un des constructeurs de sa classe pour lui donner (affecter) une valeur.

3.2.1 Attributs de validation

Les attributs de validation associés à un paramètre de fonction ou à une variable, peuvent également l'être à une propriété :

```
Class Test {
 [ValidateNotNullOrEmpty()] [string] $Property='Test'
}
$o=[Test]::new()
```

La validation se déclenchera lors de l'attribution, nommée aussi affectation, d'une nouvelle valeur :

```
$o.Test=[String]::Empty #ou ''
Exception setting "Property": "The argument is null or empty.
```

3.2.2 Hidden

Le mot clé **Hidden** masque une propriété d'une classe lors de l'énumération de ses membres :

```
Class Test {
 [string] $Property
 Hidden [int] $Compteur
}
$O=[Test]::new()
$O|Get-Member -MemberType Property
   TypeName: Test
Name     MemberType Definition
----     ---------- ----------
Property Property   string Property {get;set;}
```

L'affichage par défaut exclut les propriétés masquées :

```
$O
Property
--------
```

Attention, ce n'est pas un membre d'accès privé, la manipulation d'une telle propriété reste possible :

```
$O.Compteur=10
$O.Compteur
10
```

On doit utiliser le switch *–Force* pour le découvrir :

```
$O|Get-Member -MemberType Property -force
…
Compteur Property   int Compteur {get;set;}
Property Property   string Property {get;set;}
```

Ceci évite de lister ces propriétés lors de l'appel à l'IntelliSense dans ISE ou lors de la complétion dans une console. En C# ce comportement est implémenté via un nouvel attribut :

```
[Test].GetMember('Compteur')
MemberType       : Property
Name             : Compteur
…
CustomAttributes : {[System.Management.Automation.HiddenAttribute()]}
```

Pour le moment il n'y a pas de possibilité de créer des membres d'accès privé. Notez que la présence de cet attribut se propage dans l'objet adapté, c'est-à-dire le PSObject :

```
$O.psobject.Properties|select name
Name
----
Property
```

On ne trouve qu'une propriété, par contre on retrouve tous les accesseurs :

```
$O.psobject.Methods|select name
Name
----
get_Property
set_Property
get_Compteur
set_Compteur
…
```

Note : Les parties get et set d'une propriété s'appellent des accesseurs.

3.3 Méthode

Une méthode contient du code, sa déclaration est similaire à une fonction et à la différence que sa déclaration doit préciser, avant son nom, un type de retour ou le type **[void]** si elle ne renvoie aucun résultat :

```
Class TestMethod {
 [void] Show() {
   Write-Host "Méthode Show"
 }
}

$o=[TestMethod]::new()
$o.Show()
Méthode Show
```

Par défaut une méthode ne renvoie aucun résultat, le mot clé **[void]** est donc optionnel, dans cas la présence d'une instruction *Return* provoquera une erreur :

```
Show() {
 Write-Host "Méthode Show"
 Return "Show"
 }
Invalid return statement within void method.
```

3.3.1 $this

Dans le contexte d'une méthode d'instance l'accès à l'objet en cours se fait via le mot clés ***$this***

```
Class TestMethod {
 [string] $Path=$pwd

 [void] Show() {
   write-host "Méthode Show : $($this.Path) "
 }
}
$o=[TestMethod]::new()
$o.Show()
Méthode Show : C:\Temp
```

La propriété étant de type String on utilise la syntaxe habituelle.

Selon le contexte la variable ***$this*** référencera l'instance soit une instance, un objet, soit la classe de l'objet.

3.3.2 Paramètre

La déclaration de paramètre d'une méthode est identique à celle d'un code Powershell :

```
Class TestMethod {
 [string] $Path=$pwd

 [String] Show( [string] $Path ) {
  write-host "Méthode Show"
  Return "$($This.Path) : $path "
 }
}

$o=[TestMethod]::new()
$o.Show('C:\Windows')
Méthode Show
C:\Temp : C:\Windows
```

L'usage du mot clé **$This** permet de différencier la propriété du nom de paramètre.

L'ajout d'attributs est autorisé sur un paramètre :

```
[String] Show(
    [ValidateNotNullOrEmpty()]
    [string] $Path
 ){
  write-host "Méthode Show"
  Return "$($This.Path) : $path "
 }
```

Une évidence à rappeler, un paramètre ne peut être masqué via le mot clé **hidden**.

3.3.2.1 Nombre variable de paramètres

Pour le moment cette possibilité n'est pas implémentée via un mot clé. L'usage d'un paramètre nommé *$args* ou $*params*, ne change rien :

```
Class TestMethod {
 [string] $Path=$pwd
 [String] Params( $args ) {
   Return "Méthode Params"
 }
}
```

Si on ne précise pas le type, le paramètre est du type **Object** :

```
[TestMethod].GetMember('Params')[0].tostring()
System.String Params(System.Object)
```

On peut utiliser un tableau d'objet ou une hashtable, bref un cas à étudier…

3.3.2.2 Paramètre optionnel

Certains paramètres d'une classe C# peuvent être optionnels, par exemple :

```
Add-Type @"
 //https://msdn.microsoft.com/fr-fr/library/vstudio/dd264739%28v=vs.100%29.aspx
public class ExampleClass
{
 private string _name;
 public ExampleClass(string name = "Default name")
 { _name = name; }

  public void ExampleMethod(int required,
                            string optionalstr = "default string",
                            int optionalint = 10)
  {
    System.Console.WriteLine("{0}: {1}, {2}, and {3}.",
                             _name,
                              required,
                              optionalstr,
                              optionalint);
 }
}
"@
$Csharp=[ExampleClass]::new()
```

On peut choisir de ne pas préciser tous les paramètres lors de l'appel de la méthode :

```
$Csharp.ExampleMethod(0)
Default name: 0, default string, and 10.
```

Essayons en Powershell :

```
class TestOptional {
  [void] Optional(
     [Int] $Compteur,
     [String] $Property='Default',
     [Boolean] $Force=$false
   ){
     Write-Host "Compteur=$Compteur"
     Write-Host "Property=$Property"
     Write-Host "Force=$Force"
  }
}
$O=[TestOptional]::new() ; $O.Optional(0)
Cannot find an overload for "Optional" and the argument count: "1".
```

Vérifions le détail d'implémentation (la mise en œuvre) des paramètres de la méthode :

```
[TestOptional].GetMember('Optional').GetParameters()|
 Select Name,IsOptional,HasDefaultValue,DefaultValue
Name     IsOptional HasDefaultValue DefaultValue
----     ---------- -------------- ------------
Compteur      False           False
Property      False           False
Force         False           False
```

Tous les paramètres sont obligatoires, là où ceux d'une classe C# compilée sont optionnels :

```
[ExampleClass].GetMember('Optional').GetParameters()|
 Select Name,IsOptional,HasDefaultValue,DefaultValue
Name        IsOptional HasDefaultValue DefaultValue
----        ---------- -------------- ------------
required         False           False
optionalstr       True            True default string
optionalint       True            True 10
```

Cette fonctionnalité est prévue, mais n'est pas encore implémentée.

3.3.3 Valeur de retour

Si on remplace l'appel du cmdlet **Write-Host** par **Write-Output :**

```
Class TestMethod {
 [string] $Path=$pwd

 [String] Show() {
   Write-OutPut "Méthode Show"
   Return "Path=$($this.Path) "
 }
}
$o=[TestMethod]::new()
```

La création de la classe ne déclenche pas d'erreur, ni l'appel de la méthode, mais elle n'émet pas d'objet dans le pipeline :

```
$o.Show()|Foreach-Object {"PipeLine : $_"}
PipeLine : Path=C:\Temp
```

Seul le mot clé **Return** renvoie une valeur de retour. Celui-ci devient donc obligatoire dès lors que l'on précise une valeur de retour d'un type différent de **[void]**, son absence provoque l'erreur suivante :

```
Not all code path returns value within method.
```

Et sa présence dans une méthode précisant une valeur de retour **[void]** est interdite **:**

```
Invalid return statement within void method.
```

Attention, le code suivant est considéré comme invalide et déclenche une erreur :

```
[String] Show() {
   Write-OutPut "Méthode Show"
  if ($true) {Return "Path=$($This.Path)" }
}
Not all code path returns value within method.
```

L'ajout de l'instruction ***Else*** règle ici ce problème :

```
[String] Show() {
  Write-OutPut "Méthode Show"
  if ($true) {Return "Path=$($This.Path)" }
  else  {Return "Path=$($This.Path)" }
}
```

Enfin la valeur de retour peut être convertie implicitement dans le type attendu :

```
  [String] Show( [string] $Path,[System.Io.FileInfo] $File ) {
   write-host "Méthode Show 2"
   Return $File
  }
```

On retrouve le comportement de Powershell, car ce code appel la méthode *ToString()*. Dans le cas contraire une exception sera déclenchée :

```
[int] Show( [string] $Path,[System.Io.FileInfo] $File ) {
 write-host "Méthode Show 2"
 Return $File
}
$Result=$o.Show(1,$f[-1])
Méthode Show 2
Exception calling "Show" with "2" argument(s): "Cannot convert the
"Test.ps1" value of type "System.IO.FileInfo" to type "System.Int32"."
```

Dans ce cas, si la variable devant recevoir le résultat n'existe pas, elle n'est pas créée.

3.3.4 Hidden

Comme pour une propriété il est possible de masquer une méthode :

```
Class Test {
 [string] $Property

 Hidden [String] Show() {   Return "Méthode Show 1"  }
}
$O=[Test]::new()
$O|Get-Member -MemberType Method –force
```

```
   TypeName: Test
Name         MemberType Definition
----         ---------- ----------
Equals       Method     bool Equals(System.Object obj) ...
Show         Method     string Show() ...
```

3.3.5 Usage de variable Powershell

Par défaut les variables de la session ne sont pas accessibles dans la portée de la définition de la classe :

```
$FileName='explorer.exe'
Class TestMethod {

 [void] Show([string]$Path) {
  write-host "Méthode Show $Path\$FileName"
 }
}

$o=[TestMethod]::new()
$o.Show('C:\Windows\')
```

```
Variable is not assigned in the method.
+ FullyQualifiedErrorId : VariableNotLocal
```

Il faut préciser le modificateur de portée *global* ou *script* :

```
write-host "Méthode Show $Path\$global:FileName"
```

Toutefois évitez cette approche car la compilation ne copie pas le contenu de la chaîne, le code référencera toujours la variable globale, son contenu impactera le résultat et/ou le comportement de la méthode.

L'ajout, dans la déclaration de la méthode, d'un second paramètre est recommandé. Pour une propriété, l'exemple suivant copie la valeur de la variable :

```
$FileName=Dir
Class TestMethod {
  [string[]] $Name=$global:Filename
  [void] Show() {  write-host "Méthode Show $($this.Name)" }
}
$o=[TestMethod]::new()
$o.Show()
Méthode Show  F1.txt F2.txt F3.ps1
$FileName='Nouveau'
$o.Show()
Méthode Show  F1.txt F2.txt F3.ps1
Remove-Variable Filename
$o.Show()
Méthode Show  F1.txt F2.txt F3.ps1
```

Il est toutefois recommandé d'utiliser les paramètres d'un constructeur pour initialiser les propriétés d'une instance.

3.3.6 Surcharge de méthode

Sous Powershell la redéclaration d'une fonction écrase et remplace la précédente, il n'existe qu'une déclaration de fonction à un moment *t*. Une fonction avancée permet, au travers de jeux de paramètres, d'implémenter plusieurs comportements. La présence d'un paramètre unique dans chaque jeu déterminera le comportement attendu.

Les classes dot Net autorisent la surcharge de méthode (*method overloading*), ce principe permet à une classe de déclarer plusieurs méthodes portant le même nom, mais avec un nombre ou un type de paramètre différent, c'est-à-dire une signature d'appel différente.

Une signature étant :

Un ensemble de traits caractéristiques et reconnaissables permettant d'attribuer quelque chose à quelque chose ou à quelqu'un.

```
$S='test'
$S.Split
OverloadDefinitions
-------------------
string[] Split(Params char[] separator)
string[] Split(char[] separator, int count)
string[] Split(char[] separator, System.StringSplitOptions options)
…
```

Dans cet exemple la classe déclare la méthode *Show* quatre fois :

```
Class TestMethod {
  [string] $Path=$pwd

  [String] Show() { Return "Méthode Show 1"  }

  [String] Show( [string] $Path ) {
   write-host "Méthode Show 2"
   Return "$($This.Path) : $path "
  }
  [String] Show( [string] $Path, [object] $My ) {
   write-host "Méthode Show 3"
   Return $my
  }
  [String] Show( [string] $Path, [System.Io.FileInfo] $File ) {
   write-host "Méthode Show 4"
   Return $File
  }
}
$o=[TestMethod]::new()
$o.Show
string Show()
string Show(string Path)
string Show(string Path, System.Object My)
string Show(string Path, System.IO.FileInfo File)
```

Chaque déclaration à une signature unique, les deux dernières le sont *via* le type du deuxième paramètre.

Exemple d'appel :

```
$o.Show()
Méthode Show 1
$o.Show('MonChemin')
Méthode Show 2
C:\Temp : MonChemin
$o.Show('path',10)
Méthode Show 3
10
$Files=dir
$o.Show('path',$Files[-1])
Méthode Show 4
NomDeFichier.txt
```

Pour les deux derniers appels, le type du second paramètre détermine la méthode à appeler.

Pour l'appel suivant il n'est plus possible de déterminer le type :

```
$o.Show('path',$null)
Méthode Show 3
```

Powershell convertie $Null en un entier et recherche la méthode la plus appropriée. Si on utilise une variable typée cela ne change rien :

```
[System.IO.FileInfo] $File=$null
$o.Show('path',$File)
Méthode Show 3
```

L'appel se fait également sur la première signature. On peut forcer l'appel *via* un cast :

```
$o.Show('path',([System.IO.FileInfo]$File))
$o.Show('path',([System.IO.FileInfo]$null))
Méthode Show 4
```

La section *Multiple parameters* de cette page aborde le cas de signatures ambiguës pour du code C#, sachez qu'il existe une différence entre le CLS et le compilateur C#, celui-ci peut ajouter des contrôles. Sous Powershell, à la différence du C#, le code similaire ne pose pas de problème :

```
class Test {
  [void] Foo([int] $x, [double] $y)
  {  Write-host "Foo(int x, double y)"  }

  [void] Foo([double] $x, [int] $y)
  {  Write-host "Foo(double x, int y)"  }
}
$o=[Test]::new()
Remove-Variable a,b -ea SilentlyContinue
$a=$b=0
$o.foo($a,$b)
Foo(int x, double y)
Remove-Variable a
$o.foo($a,$b)
Foo(double x, int y)
Remove-Variable b
[double]$b=0
$o.foo($a,$b)
Foo(int x, double y)
```

Il faut juste connaitre les règles de conversions :-) ou ajouter son propre contrôle.

Le comportement suivant est pour le moins étrange :

```
$o.foo($null,$null)
Foo(double x, int y)
$o.foo(0,0)
Foo(int x, double y)
```

Par contre les appels suivants avec des variables typées fonctionnent :

```
[int]$i=$null; [int]$j=$null; $o.foo($i,$j)
Foo(int x, double y)
[double]$i=$null; [double]$j=$null; $o.foo($i,$j)
Foo(double x, int y)
```

Je suppose que la conversion diffère entre un type valeur et un type ou référence …

```
[Object]$i=$null; [Object]$j=$null; $o.foo($i,$j)
Foo(double x, int y)
```

Attention l'ordre de déclaration des surcharges pose problème, consultez ce bug pour les détails.

4 Enum

Le mot clé **enum** crée une énumération :

```
enum Drive {
 CDROM
 HardDisk
 NetWork
}
```

C'est un ensemble de constante, dont la première valeur débute par défaut à zéro :

```
[Drive]::CdRom
[Drive]::CdRom.Value__
0
```

L'usage d'une énumération facilite la compréhension du code. Entre cette déclaration :

```
[Drive]$Lecteur='NetWork'
```

et celle-ci :

```
[Drive]$Lecteur=2
```

Il est difficile de savoir à quel type correspond la valeur 2.

Elle permet également de contrôler implicitement les valeurs autorisées pour un paramètre typé d'une fonction ou d'une variable typée :

```
[Drive]$Lecteur='CDrom'
$Lecteur='CDram'
Impossible de convertir la valeur «CDram» en type «Drive».
Erreur: «Impossible de faire correspondre le nom d'identificateur CDram à
un nom d'énumérateur valide. Spécifiez l'un des noms d'énumérateur
suivants et réessayez : CDROM, HardDisk, NetWork »
```

On peut donc se passer de déclarer l'attribut *ValidateSetAttribute* qui effectue le même contrôle. Enfin l'IntelliSense et la complétion propose de choisir parmi une des valeurs d'une énumération :

```
[STA] C:\Temp> Dir C:\ -ErrorAction Continue
Continue          Ignore          Inquire         SilentlyContinue  Stop            Suspend
```

Un nom de valeur ne peut pas commencer par un chiffre. On ne peut pas pour le moment spécifier le type sous-jacent de l'énumération, celui-ci sera toujours de type ***[int]***.

Il n'est pas possible de déclarer une énumération binaire ([Flags]. En revanche on peut modifier la valeur de la première constante :

```
enum Drive {
 CDROM = 3
 HardDisk
 NetWork
}
```

```
[Drive]::CdRom.Value__
3
[Drive]::hardDisk.Value__
4
```

L'affectation de valeur spécifique est également autorisée :

```
enum Drive {
 CDROM = 7
 HardDisk = 12
 NetWork = 3
}
```

Dans ce cas l'IntelliSense, présentera toujours les constantes d'après l'ordre de déclaration.

La valeur affectée ne peut pas provenir d'un appel de code, elle doit être une constante :

```
enum Drive {
 CDROM = 5+2
 HardDisk = (get-process).count
 NetWork = 3
}
```

L'expression 5+2 est autorisée, mais pas la suivante :

```
At line:3 char:13
+  HardDisk = (get-process).count
Enumerator value must be a constant value.
```

Note : le parseur est pour le moment très strict avec sa déclaration :

```
enum Test  { absent=0 ; present=1 }
Missing ';' or end-of-line in property definition.
enum Test  { absent = 0 ; present = 1 }
```

5 Compléments

Ce chapitre approfondit les points abordés dans l'introduction.

5.1 Constructeur

Comme indiqué dans les releases notes, les propriétés comportant des expressions sont initialisées avant d'exécuter le code d'un constructeur :

```
function GetProcess {
  Write-warning "Call fonction 'GetProcess'"
  Get-Process -name S*
}

Class Test {
  [string[]] $Processes=(GetProcess)
  Test() { Write-warning "Appel le constructeur par défaut" }
}
$o=[Test]::new()
AVERTISSEMENT : Call fonction 'GetProcess'
AVERTISSEMENT : Appel le constructeur par défaut
$o.Processes
System.Diagnostics.Process (smss)
System.Diagnostics.Process (spoolsv)
...
```

5.1.1 Initialiseur

Si une classe propose un constructeur par défaut, on peut alors créer une instance puis initialiser ses propriétés à l'aide d'une hashtable :

```
Class Account{
 [string] $Name
 [Int] $Age
 [String[]] $Groups=(1..5).Foreach({"Grp$_"})
}

[Account] $Compte=@{Name='Pierre';Age=30}
```

L'affectation d'une hashtable indique ici un 'initialiseur' pour les propriétés *Name* et *Age* :

```
$Compte
Name   Age Groups
----   --- ------
Pierre  30 {Grp1, Grp2, Grp3, Grp4, Grp5}
```

La variable ***$Compte*** contient une instance de la classe *[Account]*, celle-ci est créée *via* l'appel implicite du constructeur par défaut, puis les propriétés précisées dans la hashtable sont modifiées. Ceci évite la déclaration de nombreux constructeurs ou simplifie l'écriture suivante :

```
$o=[Account]::new()
$o.Name='Pierre'
$o.Age=30
```

Dans l'exemple suivant on déclare un seul constructeur, celui par défaut n'est donc plus créé :

```
Class Account{
 [string] $Name
 [Int] $Age
 [String[]] $Groups=(1..5).Foreach({"Grp$_"})
 Account([string] $name) {$this.name=$name}
}
```

L'usage de l'initialiseur n'est plus possible dans ce cas :

```
[Account] $Compte=@{'Name'='Pierre';'Age'=30}
Impossible de convertir la valeur «System.Collections.Hashtable» du type «System.Collections.Hashtable» en type «Account».
```

La solution est de déclarer explicitement dans la classe *[Account]* le constructeur par défaut :

```
Class Account{
 [string] $Name
 [Int] $Age
 [String[]] $Groups=(1..5).Foreach({"Grp$_"})
 Account() {}     # Constructeur par défaut
 Account([string] $name) {$this.name=$name}
}
```

5.1.1.1 Conversion de type

Il est possible de créer des instances en utilisant certaines conversions de type de Powershell :

```
$Constructors = {
 [timespan]10,
 [timespan]'10',
 [timespan]'0:10',
 [System.IO.FileInfo]77 | Out-Null
}
Trace-Command -Name TypeConversion -Expression $Constructors  -PSHost
```

Pour ces instructions, le résultat des traces affiche soit l'appel à une méthode *Parse()*, soit l'appel à un constructeur :

```
...
DÉBOGUER : TypeConversion Information: 0 :  Constructor result: "00:00:00.0000010".
DÉBOGUER : TypeConversion Information: 0 :  Custom type conversion.
...
DÉBOGUER : TypeConversion Information: 0 :  Parse result: 10.00:00:00
...
DÉBOGUER : TypeConversion Information: 0 :  Parse result: 00:10:00
...
DÉBOGUER : TypeConversion Information: 0 :  Converting "77" to "System.IO.FileInfo".
DÉBOGUER : TypeConversion Information: 0 :    Constructor result: "77".
```

Ce qui fait que pour le code suivant :

```
Class Test{ [System.IO.FileInfo] $File }
$o=[Test]::new()
$o.File=15
```

L'affectation de 15 à la propriété *File* ne déclenche pas une erreur de type, mais l'appel du constructeur. Cette syntaxe peut également être utilisée avec une variable typée :

```
[System.IO.FileInfo] $File=15
$File

Mode           LastWriteTime       Length Name
----           -------------       ------ ----
darhs    01/01/1601    01:00              15
```

Il faut que la classe propose au moins un constructeur ayant un seul paramètre dont le type est identique à celui de la valeur affectée, pour que ce type d'appel implicite puisse avoir lieu.

5.1.1.2 Exemple d'implémentation

Le code suivant implémente deux cas, *Parse()* et *Create()* :

```
class Test {
  [int] $Count

  Test(){ Write-verbose "New Test" }

  Test([int] $i){
     Write-verbose "New Test(int)"
     $this.Count=$i
  }
    # Recherche d'abord la méthode Parse()
  Static [Test] Parse([string] $s){
     Write-verbose "Parse"
      #Cast implicite string -> int
     return [Test]::New($s)
  }
    # Si la méthode Parse() n'existe pas
    # recherche la méthode Create()
  Static [Test] Create([string] $s) {
     Write-verbose "Create"
      #Cast implicite string -> int
     return [Test]::New($s)
  }
}
```

Créons une instance en lui passant un entier en paramètre :

```
$VerbosePreference='Continue' ; [Test]10
VERBOSE: New Test(int)
Count
-----
   10
```

La présence d'un constructeur adéquat crée bien l'instance, maintenant passons en paramètre une chaîne de caractères :

```
$s='10' ; [Test]$s
VERBOSE: Parse
VERBOSE: New Test(int)
Count
-----
   10
```

Cette fois-ci, il n'existe pas de constructeur capable d'initialiser une instance à partir d'une chaîne de caractères. La présence de la méthode *Parse()* résout ce cas.

Les appels suivants utilisent le constructeur par défaut, puis initialise les propriétés indiquées :

```
[Test]@{ Count=10 }
$o=[PSObject]@{ Count=10 }
[Test]$O
```

Notez que le code présenté ici est une ébauche, il ne contient pas de gestion d'erreur, de plus le code des méthodes *Parse()* et *Create()* ne sont pas d'un grand intérêt de par la présence du constructeur ***Test([int] $i)*** qui utilise le cast implicite du type [string] vers le type [int].

Libre à vous de reconstruire cet exemple en ne déclarant pas ce constructeur. Enfin l'usage de la syntaxe **[NomDeClasse]::New** dans le code des méthode *Parse()* et *Create()* est nécessaire, sinon on déclenche un appel récursif.

5.1.2 Statique

Un constructeur statique, ou constructeur de type, **initialise la classe** et/ou son environnement.

Il est exécuté lors du premier accès à un constructeur de sa classe ou d'un de ses membres statiques :

```
function GetProcess {
  Write-warning "Call fonction 'GetProcess'"
  get-process -name S*
}

Class Test {
 [string[]] $Processes=(GetProcess)
 Static [int] $Data=10
 Test() { Write-warning "Call .ctor" }
 static Test() {
  Write-warning "Call .cctor"
 }
}
[Test] #Accès au type
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Test                                     System.Object
 # Création au préalable du type, puis de de l'instance.
$o=[Test]::new()
WARNING: Call .cctor
WARNING: Call fonction 'GetProcess'
WARNING: Call .ctor
```

Le constructeur statique (**.cctor** ou ***.class constructor***) est exécuté une seule fois, à moins de redéclarer la classe :

```
$o=[Test]::new()
WARNING: Call fonction 'GetProcess'
WARNING: Call .ctor
```

Remarquez qu'un accès au type ne déclenche pas l'appel du constructeur statique.

Les propriétés statiques sont initialisées avant le corps d'un constructeur statique :

```
Class Test {
 Static [int] $data=$( Write-Warning "propriété static";write-output 10)
 [string[]] $Processes=(GetProcess)

 static Test() {  Write-warning "Call .cctor" }
 Test() { Write-warning "Call .ctor" }
}
[Test]::Data
WARNING: propriété static
WARNING: Call .cctor
10
[Test]::Data
10
```

5.1.3 Exception

Une exception dans un constructeur d'instance renvoie ***$null***.

Une exception dans un constructeur statique invalide la classe :

```
Class Test {
 static [string] $Str='Test'
 static Test() {
  Write-warning "Call .cctor"
  1/0
 }
}
$O=[Test]::new()
WARNING: Call .cctor
Exception calling ".ctor" with "0" argument(s): "The type initializer for
'Test' threw an exception."
$null -eq $O
True
[Test]::Str
The type initializer for 'Test' threw an exception.
```

Dans le cas où la variable **$O** existe avant l'appel, son contenu n'est pas modifié.

Note :

Pour le moment, un constructeur ne peut appeler un autre constructeur du même objet à l'aide du mot clé ***$this*** :

```
public Employee([int] $annualSalary)
{ … }
public Employee([int] $weeklySalary, [int] numberOfWeeks)
 :$this($weeklySalary * $numberOfWeeks)  #Pas supporté
{ …}
```

L'usage d'une méthode intermédiaire est nécessaire.

5.2 Propriété

5.2.1 Attributs de validation

Nous avons vu que les propriétés comportant des expressions sont initialisées avant d'exécuter le code d'un constructeur. Les propriétés suivantes sont initialisées, mais pas validées car la validation se déclenche uniquement lors de l'affectation :

```
Class Test {
  [ValidateNotNullOrEmpty()] [string] $Property
  [ValidateRange(5,10)][int] $Compteur
}
$o=[Test]::new()
$o
```

L'appel du constructeur par défaut réussie. Mais il échouera si on affecte une valeur invalide qui peut être une constante ou le résultat d'une expression (fonction, cmdlet, méthode statique,…) :

```
Class Test {
  [ValidateNotNullOrEmpty()] [string] $Property
  [ValidateRange(5,10)][int] $Compteur= ((Get-Process).Count)
}
$o=[Test]::new()
Exception calling ".ctor" with "0" argument(s): "The 49 argument is less
than the minimum allowed range of 5. …
```

On peut envisager de forcer la validation :

```
Class Test {
 [ValidateNotNullOrEmpty()] [string] $Property
 [ValidateRange(5,10)][int] $Compteur

 Test () { $this.Validate() }

 Hidden [void] Validate(){
   Write-Warning "Validate"
   $this.Property=$this.Property
   $this.Compteur=$this.Compteur
 }
}
$o=[Test]::new()
Exception calling "Validate" with "0" argument(s): "Exception setting
"Property": "The argument is null or empty….
```

Note : Certains attributs du Framework destinés à un compilateur, par exemple *System.Obsolete*, ne sont pas pris en charge.

5.2.2 Statique

Le mot clé **Static** déclare une propriété statique, c'est-à-dire que l'appel au constructeur n'est pas nécessaire pour la manipuler :

```
Class Test {
  static [string] $Path='C:\temp'

  [void] Show() {  write-host "Méthode Show '$( $this::Path )'" }
}
```

Une propriété statique est accessible depuis le nom de classe uniquement. Pour manipuler un membre statique l'usage des deux points '**::**' est nécessaire :

[NomDeClasse]**::***NomDePropriété*

Par exemple :

```
[Test]::Path
C:\temp
```

La propriété statique *Path* n'existe pas sur une instance de la classe ***[Test]*** :

```
$o=[Test]::new() #Crée une instance
$o.Path
#ras
$o.Show()
Méthode Show C:\temp
```

Une méthode d'instance peut accéder à une propriété statique de sa classe. Notez que dans le code de la méthode *Show()*, la syntaxe **$this::Path** permet de référencer la classe en cours de déclaration. Le code suivant est identique :

```
[void] Show() { write-host "Méthode Show '$( [Test]::Path )'" }
```

La propriété *Path* étant partagée par toutes les instances de la classe *[Test]*, sa modification concerne donc toutes les instances :

```
[Test]::Path='C:\Windows'
$o=[Test]::new()
$o2=[Test]::new()
$o.Show()
Méthode Show C:\Windows
$o2.Show()
Méthode Show C:\Windows
```

Par défaut **Get-Member** n'affiche pas les membres statiques d'un objet ou d'une classe :

```
$o|Get-Member -MemberType property
# ras
```

On doit préciser son paramètre –*Static* :

```
$o|Get-Member -MemberType property -static
   TypeName: Test
Name MemberType Definition
---- ---------- ----------
Path Property   static string Path {get;set;}
```

L'appel suivant est similaire :

```
[Test]|Get-Member -MemberType property -Static
```

5.2.3 Valeur par défaut

Comme dit précédemment, une propriété peut définir une valeur par défaut, sinon elle reçoit celle du type :

```
Class ValeurParDefaut {
 [string] $Property='Test'
 [string] $Defaut
 [string] $PropertyNull=$Null
 [string]  $NullString= [System.Management.Automation.Language.NullString]::Value
 [Int]     $Number=$Null

 [System.IO.FileInfo] $Config=$null
 [System.IO.FileInfo] $File=10
}
$o=[ValeurParDefaut]::new()
```

On constate que l'affectation de *$null* à une propriété de type string, vaut une chaîne de caractère vide :

```
$o.PropertyNull -eq [string]::empty
True
```

Ceci est normal puisque cette déclaration n'est pas transformée en un code C# avant la compilation, Powershell exécute l'expression puis affecte le résultat. La variable **$Number** contient zéro et l'affectation de *$null* ne déclenche pas d'exception.

En revanche la valeur par défaut d'une propriété de type string vaut, comme en C#, *$null* :

```
$null -eq $o.Defaut
True
```

Une règle de plus à connaitre.

Dans l'exemple suivant, se basant sur la conversion de type vu précédemment, le constructeur appelé dépend du type de la valeur affectée :

```
Class Account{
 [string] $Name
 [Int] $Age

 Account( [string]$Name ) {
  Write-warning "Call .ctor [String]"
  $this.Name=$Name
 }
 Account( [int]$Age ) {
  Write-warning "Call .ctor [Int]"
  $this.Age=$Age
 }
}
[Account] $Compte=10
AVERTISSEMENT : Call .ctor [Int]
$Compte
Name   Age
----   ---
        10
[Account] $Compte='Pierre'
AVERTISSEMENT : Call .ctor [String]
$Compte
Name   Age
----   ---
Pierre   0
```

Attention aux déclarations suivantes :

```
Class Test{ [System.IO.FileInfo] $File=10 }
$o=[Test]::new()
$o.File|select name,mode
Name Mode
---- ----
10   darhsl
```

Le code suivant est différent, bien que l'affichage du résultat soit identique :

```
Class Test{  $File= [System.IO.FileInfo]10 }
$o2=[Test]::new()
$o2.File|select name,mode
Name Mode
---- ----
10   darhsl
```

Dans ce cas Powershell crée la propriété *File* avec le type **System.Object**, elle peut donc contenir n'importe quel type d'objet :

```
$o2.File=Get-Process
```

Pour la première syntaxe, la même affectation déclenchera une exception :

```
$o.File=Get-Process
Exception setting "File": "Cannot convert the "System.Object[]" value of
type "System.Object[]" to type
"System.IO.FileInfo"."
```

5.2.4 Scope

Dans l'exemple suivant, la définition du membre statique *$Count* utilise la variable **$D** uniquement lors de la construction de l'assembly dynamique, on pourrait ici parler de compilation, mais je fais le choix de considérer Powershell comme un langage dynamique de 'bout en bout' :

```
$D=99
class Test
{
  $Count = $D
  Static $B=$D
}
[Test]::B
99
$D=10
```

Lors de l'appel du constructeur, la définition du membre *$Count* utilise la valeur courante de la variable **$D** :

```
$o=[Test]::new()
$o.Count
10
```

Une fois créée la classe *[Test]* ou l'instance $O, la modification de la valeur de la variable **$D** ne change rien à l'existant :

```
$D=15
[Test]::B
99
$o.Count
10
```

5.2.5 Accesseurs

En interne une propriété est constituée d'un champ et de deux méthodes, un getter et un setter. Leurs noms sont respectivement **get_***NomDePropriété* et **set_** *NomDePropriété.*

Une propriété joue le rôle d'un intermédiaire afin de contrôler les accès au champ associé. L'accesseur *Set* peut être utilisé pour implémenter des règles de gestion. Malheureusement la version 5 actuelle ne permet pas de déclarer des champs, mais uniquement des propriétés.

La redéclaration des accesseurs d'une propriété est possible :

```
class Test
{
  $Count

  [System.Object] get_Count(){
    Write-warning "Get $($this.count)"
    return $this.count
  }

  [void] set_Count([System.Object] $value){
    Write-warning "Set $value)"
    $this.count=$value
  }
}
```

Mais cette approche ne fonctionne pas pour cet exemple, ces méthodes sont ajoutées à la liste des membres sans être associées en interne à la propriété *Test.Count*.

De plus, comme le champ devant être associé à la propriété n'est pas accessible, le code suivant de la méthode *set_Count*, provoquerait un appel récursif :

```
$this.count=$value
```

L'usage de l'attribut ***ValidateScript*** aurait pu être une solution, mais celui-ci ne fonctionne pas.

Nous verrons plus avant qu'il existe un contournement, mais par défaut on ne peut déclencher l'exécution de code lors de l'accès à une propriété. Cette possibilité sera implémentée dans une prochaine version.

5.3 Méthode

Une méthode contient du code et sa déclaration doit préciser, à la différence d'un code Powershell, un type de retour ou le type **[void]** si elle ne renvoie aucun résultat :

```
Class TestMethod {
 [string] $Path
 [void] Test () {  write-host "Méthode Test"  }
}
```

Une méthode ne propose pas les blocs *begin*, *end* et *process*, car ce n'est ni un cmdlet ni une fonction avancée.

5.3.1 Statique

Le mot clé **Static** concerne également les méthodes. Reprenons l'exemple utilisé pour les propriétés statiques et déclarons la méthode *Show()* en tant que statique :

```
Class Test {
  static [string] $Path='C:\temp'
  Static [void] Show() { write-host "Méthode Show '$( $This::Path )'" }
}
```

Cet exemple ne peut pas fonctionner tel quel :

```
Impossible d'accéder au membre non statique This dans une méthode statique
ou dans l'initialiseur d'une propriété statique.
```

Dans une méthode d'instance la variable *$this* référence l'instance en cours ou la classe en cours. L'ajout du mot clé **stati**c modifiant l'accès à la méthode, on doit utiliser le nom de la classe :

```
Class Test {
  static [string] $Path='C:\temp'
  Static [void] Show() { write-host "Méthode Show '$( [Test]::Path )'" }
}
[Test]::Show()
Méthode Show 'C:\temp'
```

Dans l'exemple suivant on obtient la même erreur, car la propriété *Path* n'est plus statique, ceci fait que le code de la méthode statique ne peut plus y accéder :

```
Class Test {
  [string] $Path='C:\temp'
  Static [void] Show() { write-host "Méthode Show '$( $this.Path )'" }
}
Impossible d'accéder au membre non statique This dans une méthode statique
ou dans l'initialiseur d'une propriété statique.
```

Une méthode statique ne peut donc manipuler que des propriétés statiques ou d'autres méthodes statiques de sa classe, bien évidement elle peut manipuler d'autres objets passés en paramètre :

```
Class Test {
  [string] $Path='C:\temp'
  Static [void] Show() { write-host "Méthode statique" }
}
[Test]::Show()
Méthode statique
```

5.3.2 Exception

Si, lors de l'affectation du résultat d'une méthode, une exception se déclenche et que la variable existe alors son contenu n'est pas modifié, si la variable n'existe pas elle n'est pas créée.

Par défaut une exception déclenchée par une méthode d'un objet n'est pas bloquante, le post suivant indique que la présence d'un bloc ***try/catch*** transforme l'exception en une erreur bloquante.

6 Autres aspects

6.1 Propriété de type Scriptblock

Si toutefois on souhaite affecter du code à une propriété, il faut utiliser un scriptblock :

```
$A=10
Class Test {
    [scriptblock] $Code= {
        Write-Host "Code A=$A"
    }

    Static [scriptblock] $Code2= {
        Write-Host "Code2 A=$A"
    }
}
$o=[Test]::New()
&$o.Code
Code A=10
```

Pour la propriété statique l'usage de parenthèses est nécessaire, à moins d'appeler la méthode Invoke() du scriptblock :

```
&([Test]::Code2)
[Test]::Code2.Invoke()
Code2 A=10
```

6.2 Scope

A la différence d'une fonction qui utilise la portée dynamique, une classe Powershell utilise la portée lexicale c'est-à-dire que son code référence le contexte où on la définie.

L'exemple suivant est adapté de celui cité dans la documentation en ligne *about_classes* :

```
$d = 42  # Script scope
function PSFonction
{
   $d = 0  # Function scope
   [MyClass]::DoSomething()
}

class MyClass
{
   static [object] DoSomething()
   { return $script:d }
}
```

Nous avions vue précédemment la nécessité d'utiliser le modificateur de portée, ici on constate que la variable **$D** référencée dans la méthode *DoSomething()* n'est pas celle déclarée dans la portée de la fonction *PSFonction* :

```
$v = PSFonction
$v
42
$v -eq $d
#true
```

La variable **$D** référencée est celle de la portée où l'on déclare la classe.

Dans l'exemple suivant on déclare la classe dans un script bloc, la classe référence la portée qui contient sa déclaration :

```
$d=76
$d=77

&{
  $d = 42  # Script scope

  function PSFonction
  {
     $d = 0  # Function scope
     [MyClass]::DoSomething()
  }


  class MyClass
  {
     static [object] DoSomething()
     { return $script:d }
  }
  $v = PSFonction
  "V=$v"
  $v -eq $d
  "d=$d"
}
V=77
False
d=42
```

Le parsing de ce code source fait que la classe référence ***$d=77,*** le code du scriptblock est bien vu comme une nouvelle portée, mais celle-ci existera uniquement lors de l'exécution du code et c'est à ce moment-là que ***$d*** vaudra ***42***.

Si on insère ce code dans un script, c'est la portée du script qui prime, le dotsource ne modifiera pas ce comportement puisque la classe utilise la portée lexicale.

Maintenant inversons la situation, c'est-à-dire que le code d'une méthode appelle une fonction :

```
$d = 42  # Script scope

 function PSFonction2
 {  return $d }

 class MyClass
 {
     [int] CallPSFunction2()
     {
         $d=-1
         return PSFonction2
     }
 }
PSFonction2
42
$o=[MyClass]::New()
$o.CallPSFunction2()
-1
```

Dans ce cas la fonction utilise la portée dynamique, ce qui est normal, mais sujet à confusion. Ici on ne fait qu'empiler ou combiner différents contextes d'exécution.

Bien que ceci soit possible, je doute que ce type de codage soit à promouvoir. En passant, le code de la fonction doit impérativement respecter le type de la valeur de retour déclarée :

```
function PSFonction2
 {
   Write-Output $d
   Write-Output $d
 }
$o.CallPSFunction2()
```

Dans le cas contraire son exécution nous le rappellera :

```
Exception calling "CallPSFunction2" with "0" argument(s): "Cannot convert
the "System.Object[]" value of type "System.Object[]" to type
"System.Int32"."
```

6.2.1 Script

Créons dans un script une classe et une fonction renvoyant une instance de cette classe, c'est-à-dire que l'on crée une instance dans un contexte de script pour l'utiliser dans un autre contexte :

```
@'
$d = 99  # Script scope
class MyClass{
  [Int] $Compteur =$script:d

  static MyClass() {
      Write-warning "Call .cctor [MyClass]"
  }

  static [object] DoSomething() { return $script:d }

  [object] getVariableValue() { return $script:d }
}

function New-MyClass {
  $o=[MyClass]::New()
  Write-host "Compteur $($o.Compteur)"
  Return $o
}

New-MyClass
'@ > C:\Temp\scope1.ps1
```

Exécutons le script et récupérons l'instance :

```
Remove-Variable D -EA SilentlyContinue
$Objet=C:\Temp\scope1.ps1
Compteur 99
$objet.Compteur
99
```

Maintenant exécutons sa méthode d'instance :

```
$objet.getVariableValue()
```

Celle-ci ne renvoie aucun résultat, créons la variable **$d** :

```
$D=10
$objet.getVariableValue()
10
```

L'instance utilise la portée courante, vérifions le contenu de la propriété *$Compteur* :

```
$objet.Compteur
99
```

Elle contient bien la valeur de la variable **$d** déclarée dans le script. Testons maintenant sa méthode de classe *DoSomething()* :

```
[MyClass]::DoSomething
Unable to find type [MyClass].
```

On constate que la classe créée dans la portée du script n'est pas accessible dans la portée courante, mais elle existe bien en mémoire :

```
[System.AppDomain]::CurrentDomain.GetAssemblies()|
 where { ($_.Location -eq $null) -and ($_.DefinedTypes.Count -ne 0)}|
 Select DefinedTypes
DefinedTypes
------------
{Type$ConvertByrefToPtr}
{MyClass, <MyClass_staticHelpers>}
```

Le nom de l'assembly référence le script où a été créée la classe :

```
$objet.GetType().Assembly.Fullname
C__Temp_scope1.ps1, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
```

Récupérons le type de notre instance appelons sa méthode de classe :

```
$objet.GetType()::DoSomething()
10
```

Remarquez que la méthode utilise la portée courante et pas celle du script, on récupère donc la valeur 10 et pas celle du script qui était 99.

Revoyons l'explication sur la portée lexicale :

```
function get { $d=-1; $objet.GetType()::DoSomething()}
get
10
```

Elle est toujours valide, le modificateur de portée *script:* référence désormais la portée courante.

Vous avez dû noter l'appel du constructeur de classe :

```
$Objet=C:\Temp\scope1.ps1
AVERTISSEMENT : Call .cctor [MyClass]
Compteur 99
```

Celui-ci est exécuté une seule fois, lors du premier appel du script, les prochains appels ne le déclencheront plus :

```
$Objet2=C:\Temp\scope1.ps1
Compteur 99
```

Les variables *$Objet* et *$Objet2* sont donc du même type.

6.2.2 Module

Les classes peuvent être déclarées dans un module :

```
@'
 [int]  $_compteur=0

 class  Test{
   [void] Inc(){ $script:_compteur++ }
   [Int] GetCompteur(){ Return $script:_compteur }
 }
'@ > c:\temp\Test.psm1
Import-Module c:\Temp\Test.psm1
```

Le code d'une classe et de ses instances référencera toujours l'état de session du module, on peut ainsi coupler une de ses variables privées à une méthode, comme on peut le voir dans la méthode *Inc()*.

Notez que ceci est différent d'une propriété privée (*private*) puisque chaque instance utilisera la même variable. Pour cette version, la limite est que les classes déclarées sont invisibles dans la portée appelant le module :

```
[Test]
Unable to find type [Test].
```

Ce qui implique de déclarer une fonction publique dédiée à la création d'instance :

```
@'
 [int]  $_compteur=0
 class  Test{
   [void] Inc(){ $script:_compteur++ }
   [Int] GetCompteur(){ Return $script:_compteur }
 }
 function New-Test() { [Test]::New() }
'@ > c:\temp\Test.psm1
Import-Module C:\Temp\Test.psm1
$o=new-Test
```

Comme pour un script, le nom de l'assembly de la classe référence le module où elle a été créée :

```
$o.gettype().Assembly.FullName
C__temp_Test.psm1, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
```

Au lieu de créer des wrappers sur ses propres classes, il reste possible d'utiliser un raccourci de type :

```
$samT='System.Management.Automation.TypeAccelerators'
$AcceleratorsType= [PSObject].Assembly.GetType($samT)
$AcceleratorsType::Add('Test',[Test])
```

L'inconvénient majeur est que si un autre module utilise le même nom de classe ou que l'on utilise le versionning de module qui permet de charger simultanément plusieurs versions d'un même module (side-by-side (SxS) module version), le dernier module chargé écrasera la définition courante.

La classe ModuleInfo propose désormais une propriété supplémentaire nommée *ImplementingAssembly*, elle référence l'assembly contenant les classes déclarées dans le module :

```
$ModuleInfo.ImplementingAssembly.DefinedTypes|Where isPublic
IsPublic IsSerial Name   BaseType
-------- -------- ----   --------
True     False    Test   System.Object
```

Celle-ci autorise une autre approche, qui est d'interroger la liste des types d'un module :

```
function Get-Class {
  [CmdletBinding()]
  param
  (
      [Parameter(Mandatory,ValueFromPipeline)]
      [System.Management.Automation.PSModuleInfo]
      $Module,

      [Parameter(Mandatory)]
      [ValidateNotNullOrEmpty()]
      [string]
      $ClassName,

      [Switch] $Strict
  )

 $Result=@(
  $Module.ImplementingAssembly.DefinedTypes|
   Where {$_.isPublic -and $_.Name -eq $ClassName}
 )

 if ($Result.Count -eq 0)
 {
   $Exception=New-Object System.ArgumentException("Type
inconnu.",'ClassName')
   If (!$Strict)
   {
    $PSCmdlet.WriteError(
```

```
      (New-Object System.Management.Automation.ErrorRecord(
        $Exception,
        "PowerShellClassNotFound",
        "ObjectNotFound",
        ("[{0}]" -f $ClassName)
       )
      )
    )
   }
   else
   { throw $Exception }
 }
 else {$Result}
}
Get-Module Test | Get-Class -ClassName 'Test'
```

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Test                                     System.Object
```

Pour des modules *side-by-side* on utilisera le paramètre ***FullyQualifiedName*** :

```
$FQN=@{ModuleName="Test"; ModuleVersion ='2.0'; GUID='c51bd812-33ca-4a5c-
bed8-7bc9ef50a9d2'}
Get-Module -FullyQualifiedName $FQN |
 Where {$_.Version -eq '2.0'}|
 Get-Class -ClassName 'Test'
```

6.2.2.1 Remove-Module

La suppression d'un module ne supprime pas l'assembly qu'il a créé :

```
$a=Get-Assemblies|
 where {($_location -eq $null) -and ($_.fullname -match '.psm1')}
$a.Modules|
 Select MDStreamVersion,ModuleVersionId
```

Les objets associés à l'état de session du module restent valides, en interne les données du module supprimé restent accessibles. Le chargement et déchargement d'un même module ne créé pas un nouvel assembly, il semble être mis en cache et sa suppression 'retardée'.

Dans le cas où on modifie la déclaration d'une classe d'un module, Powershell crée un nouvel assembly.

6.2.2.2 Constructeur statique

Comme pour un script les constructeurs statiques sont exécutés une seule fois :

```
@'
class Test {
    [int] $Count
    Test() { Write-warning "Call .ctor [Test]" }
    static Test() {
      Write-warning "Call .cctor [Test]"
    }
}
 function Get-Testtype {[Test]}
'@ > c:\temp\TestType.psm1
IPMO c:\temp\TestType.psm1
AVERTISSEMENT : Call .cctor
$type=Get-Testtype
$Old=$type::New()
AVERTISSEMENT : Call .ctor [Test]
Remove-Module TestType
IPMO c:\temp\TestType.psm1
$type=Get-Testtype
$New=$type::New()
AVERTISSEMENT : Call .ctor [Test]
```

Les instances contenues dans les variables *$Old* et *$New* sont du même type :

```
$New.getType() -eq $Old.GetType()
True
```

6.2.2.3 Déclaration avancée (Forward) ?

Pour information le code d'un module peut référencer une classe avant sa création :

```
@'
 Write-warning [Test].Assembly.Fullname

 [int] $_compteur

 class Test{  [int] $Compteur }
'@ > c:\temp\Test.psm1
Ipmo c:\temp\Test.psm1
```

Le code précédent ne provoque pas d'erreur, l'appel d'une méthode statique est également possible. La contrainte est que son code doit pouvoir accéder à toutes les ressources qu'elle référence. Par exemple une telle méthode ne peut appeler une fonction qui serait déclarée après le code de sa propre déclaration.

Ce comportement est dû je pense au fait que l'assembly est créé avant l'exécution du code du module.

6.2.2.4 Toc Toc Toc ! Qui est là ?

Certains cas peuvent déclencher un message similaire à celui-ci où *Test* est un nom de classe :

```
Impossible de convertir la valeur « Test » du type « Test » en type
« Test ».
```

Ce message d'erreur légèrement déroutant se produit dans le cas où une variable typée se voit affecter une instance d'une classe ayant la même définition, mais hébergée dans un assembly dynamique différent. Par exemple avec deux versions de modules côte à côte.

Un exemple où la variable **$A** est contrainte sur le type *[Test]* :

```
class Test
{
    [int] $Count
    Test(){ Write-verbose "New Test()" }
}
[Test]$A=[Test]@{Count=10}
```

La redéfinition de la classe doit se faire en deux passes sinon Powershell déclenche l'erreur :

```
Le membre Test est déjà défini.
```

On redéfinie la classe en la modifiant ce qui force Powershell à créer un nouvel assembly :

```
class Test
{
    [int] $Count
    [string] $Msg
    Test(){ Write-verbose "New Test()" }
}
```

Puis on réutilise la variable $A en lui affectant une instance créée avec la nouvelle définition de la classe, de même nom :

```
$A=[Test]@{Count=10}
Impossible de convertir la valeur « Test » du type « Test » en type
« Test ».
```

Si dans la console on exécute deux fois la même déclaration, l'erreur ne se produit pas, car dans ce cas Powershell utilisera le même assembly présent dans le cache interne.

6.3 Héritage

Rapidement, ce concept sert à regrouper, selon des caractéristiques communes, des classes au sein d'une hiérarchie, L'héritage permet également de réutiliser du code et de spécialiser des classes.

L'administration système utilise également ce concept, par exemple avec l'héritage de droits NTFS ou AD, dans notre cas nous allons, non plus le paramétrer, mais le concevoir.

L'héritage propose aux classes descendantes de bénéficier des caractéristiques de leurs ancêtres, à savoir leurs propriétés et méthodes. On déclare l'héritage sur une classe enfant en précisant son parent :

```
class Enfant : Parent
```

On utilise également les termes correspondant suivants :

```
class  SousClasse : Superclasse
```

Voir ceux-ci :

```
ClasseDérivée :ClasseDeBase
```

Les deux noms de classe sont séparés par le caractère deux points. Il ne peut y avoir qu'un nom de classe après ce caractère, on peut toutefois ajouter un ou des noms d'interfaces séparés par des virgules :

```
class Component : MarshalByRefObject,IComponent,IDisposable
```

Par convention un nom d'interface débute par le caractère '**I'**

Ici *MarshalByRefObject* est un nom de classe, *IComponent* et *IDisposable* des noms d'interfaces.

Une classe enfant peut implémenter une interface sans hériter d'une classe :

```
class Enfant : IDisposable
```

Pour le moment sachez qu'ici une interface ajoute un comportement que la classe est obligée d'implémenter, l'interface porte la responsabilité, pas le piano…

Créons deux classes, d'abord la classe *[Parent]* :

```
class Parent {
    [string] $Nom
    [string] $Prenom
    [byte] $Age

    Parent(  [string] $Nom,  [string] $Prenom) {
       $this.Nom=$Nom
       $this.Prenom=$Prenom
    }
}
```

Puis la classe *[Enfant]* :

```
class Enfant : Parent { }
```

La classe nommée *Enfant* ne déclare pas de membre, mais hérite de ceux de la classe ancêtre, SAUF le constructeur. Ce qui fait que la création de la classe *[Enfant]* échoue :

```
Base class 'Parent' does not contain a parameterless constructor.
```

On doit déclarer un constructeur proposant au moins les deux paramètres présent dans le constructeur de la classe [Parent] :

```
class Enfant : Parent
{
   Enfant ([string] $Nom,[string] $Prenom):base($Nom,$Prenom){}
}
```

Puisque la classe *[Enfant]* ne déclare pas de membre, mais hérite ceux de la classe ancêtre, c'est celle-ci qui doit réserver l'espace mémoire dédié au contenu des propriétés. C'est la classe ancêtre qui connait le nombre et le type de ses propriétés. Pour que l'enfant existe, il faut, en interne, d'abord créer le parent :

```
$o=[Enfant]::New('Dupond','Abdel')
$o

Nom    Prenom Age
---    ------ ---
Dupond Abdel    0
```

Dans un héritage, on doit appeler en cascade au moins un constructeur de chaque ancêtre, s'il en existe plusieurs c'est à vous de décider lequel appeler. Chaque constructeur d'une classe dérivée n'appel que le constructeur de son parent.

6.3.1 Appel d'un constructeur d'une classe de base

Pour signaler que l'on enchaine un appel de constructeur, on utilise le mot clé '**base**' :

```
class Enfant : Parent
{
   Enfant ([string] $Nom,[string] $Prenom):base($Nom,$Prenom){}
}
```

On reprend la syntaxe de déclaration de l'héritage de classe, mais cette fois-ci on manipule une signature de méthode d'un constructeur.

La syntaxe du mot clé '**base**' est identique à un appel de méthode :

```
base($Nom,$Prenom)
```

Les deux paramètres sont ceux utilisés dans la déclaration du constructeur de la classe *[Parent]* :

```
Parent(  [string] $Nom,  [string] $Prenom)
```

Leur contenu est automatiquement affecté lors de l'appel du constructeur de la classe *[Enfant]*.

L'opération effectuée correspond plus ou moins à ceci :

```
[Enfant]::New('Dupond','Abdel') {
  $Instance=[Parent]::New('Dupond','Abdel') # base
 Return $Instance –as [Enfant]
}
```

Par convention on utilise les mêmes noms de paramètres. Leur nombre peut varier sur le constructeur de la classe dérivée, ici *[Enfant]* :

```
Enfant ([string] $Nom):base($Nom,'AbDel'){}
```

Mais il ne peut pas varier sur notre constructeur de la classe *[Parent]*, à moins qu'elle en propose d'autres.

Il reste possible d'ajouter des paramètres au constructeur de la classe *[Enfant]* :

```
class Enfant : Parent
{
   [String] $Adresse

   Enfant ([string] $Nom, [string] $Prenom, [String] $Adresse):
      base($Nom,$Prenom){
    $this.Adresse=$Adresse
   }
}
```

L'appel du constructeur parent ne change pas, puisque seule la classe *[Enfant]* déclare la propriété *Adresse*.

Dans le cas suivant, les constructeurs par défaut de chaque classe sont appelés automatiquement :

```
class Parent
{
    [string] $Nom
    [string] $Prenom
    [byte] $Age
}

class Enfant : Parent { }
```

La solution à notre premier exemple est d'ajouter un constructeur par défaut :

```
class Parent
{
    [string] $Nom
    [string] $Prenom
    [byte] $Age

    Parent() {}
```

```
    Parent(
      [string] $Nom,
      [string] $Prenom) {

      $this.Nom=$Nom
      $this.Prenom=$Prenom
    }
}

class Enfant : Parent { }
```

Dans ce cas l'enchainement des constructeurs par défaut peut se faire. En dehors de ces problèmes de mécanique, on peut se poser la question suivante :

Doit-on autoriser la création d'une instance d'une de ces classes avec les propriétés 'Nom' et 'Prénom' avec une chaîne de caractères vide ?

Note : J'ai choisi d'utiliser les noms de classe '**Parent**' et '**Enfant**' bien qu'ils puissent induire en erreur, notamment sur la relation entre leurs instances. Ici il n'y a pas de notion de *frère* ou *mère*.

Deux instances d'une classe *[Personne]* seraient plus appropriées pour porter une relation de filiation. Il faut donc dire lire 'ClasseParente' et 'ClasseEnfant', ou mieux les renommer en :

```
class ClasseDeBase{}

class ClasseDérivée : ClasseDeBase {}
```

6.3.2 Substitution de méthode

Si l'héritage permet de spécialiser une classe, nous devons remplacer un traitement partagé par un traitement spécifique à la classe. La substitution de méthode (*method overriding*) permet cette modification.

```
class ClasseDeBase{
    [int] $Compteur=1
    [int] $Increment

    ClasseDeBase() { Write-Warning "`tnew ClasseDeBase" }

    [string] ToString(){
      Return "traitement de la classe de base. Increment=$($this.Increment)"
    }

    [int] Compter(){
      Write-Warning "$this"
      Write-Warning $this.GetType().FullName
```

```
      Return ($this.Compteur * $this.Increment)
    }
}

class ClasseDérivée  : ClasseDeBase {
  [int] $Increment= 10

   ClasseDérivée() { Write-Warning "new ClasseDérivée"  }

  [string] ToString(){
    Return "traitement de la classe dérivée. Increment=$($this.Increment)"
  }
}
```

Notez que chaque déclaration de constructeur ne modifie aucune propriété :

```
$Base=[ClasseDeBase]::new()
$Dérivée=[ClasseDérivée]::new()
```

Bien qu'elle ne déclare pas explicitement la méthode *Compter ()*, la classe **ClasseDérivée** possède automatiquement cette méthode, car elle hérite des membres de la classe de base ***:***

```
$Dérivée|Get-Member
…
Compter      Method       int Compter() …
```

Alors qu'il n'existe qu'une seule déclaration, on constate que le résultat renvoyé diffère pour les deux variables :

```
$Base.Compter()
WARNING: traitement de la classe de base. Increment=0
WARNING: ClasseDeBase
0
$Dérivée.Compter()
WARNING: traitement de la classe dérivée. Increment=10
WARNING: ClasseDérivée
10
```

Ceci est dû au fait que la variable **$this** référence l'instance qui a déclenchée l'appel et pas la classe dans laquelle elle est déclarée.

Dit autrement, lors de l'exécution de la méthode *Compter()* les occurrences de **$this** sont 'remplacées' en interne par la variable à l'origine de l'appel.

L'instruction suivante appelle indirectement la méthode *ToString()* :

```
      Write-Warning "$this"
```

Comme celle-ci est redéfinie dans chaque classe, le message s'adapte au type de l'instance.

Une fois une méthode substituée, il reste possible d'appeler celle de la classe de base. On doit caster l'instance, en français la transtyper, c'est dire informer le compilateur/parseur que l'on considère l'instance comme étant d'un de ces types ancêtre.

Dans l'exemple suivant la variable *$Dérivée* est précédée du nom de son type ancêtre et cette expression doit être entre parenthèses :

```
([ClasseDeBase]$Dérivée).tostring()
traitement de la classe de base. Increment=10
$Dérivée.ToString()
traitement de la classe dérivée. Increment=10
```

Dans celui-ci, le cast est redondant puisqu'il n'existe qu'une méthode *Compter()* :

```
([ClasseDeBase]$Dérivée).Compter()
WARNING: traitement de la classe dérivée. Increment=10
WARNING: ClasseDérivée
10
```

Il n'existe pas de mot pour préciser une redéfinition de méthode, comme *Override* ('primer sur' en anglais). Toutes les méthodes sont virtuelles, la présence d'une redéfinition de méthode ayant le même nom et la même signature suffit. Ceci est également valable pour redéfinir une méthode qui n'est pas virtuelle. Enfin, rien ne s'oppose à ce qu'une méthode virtuelle soit surchargée.

On ne peut pas dire à la seule lecture du code d'une méthode virtuelle ce qu'elle fera ou affichera, cela dépend du type de l'instance qui l'appel. Ce mécanisme permet à une classe définie dans le Framework dot Net d'appeler le code de vos méthodes sans que vous ayez quoi que soit à faire, si ce n'est lire la documentation de la classe dot Net.

6.3.3 Classe générique

Ce que nous venons de voir concerne également les classes génériques. On peut par exemple créer une nouvelle classe à partir d'une classe C# générique :

System.Collections.Generic.List<T>

Ici la syntaxe **<*T*>** indique que l'on doit préciser un nom de classe, par exemple ***Int.*** En quelque sorte, on paramètre une classe à l'aide d'un type, y compris une classe Powershell.

Sous Powershell on utilise une autre syntaxe pour référencer un type générique :

[System.Collections.Generic.List[T]]

```
class MyIntList : system.collections.generic.list[int]
{
    # Add is final in system.collections.generic.list
    [void] Add([int]$arg)
    {
        ([system.collections.generic.list[int]]$this).Add($arg * 2)
    }
}
```

```
$list = [MyIntList]::new()
$list.Add(100)
$list[0]
200
```

Pour l'appel d'une méthode de la classe de base on reprend le nom complet de la classe générique.

L'objet **$List** est une liste d'objet du type entier, elle ne peut contenir que des instances de la classe **int**, dans le cas contraire une exception est déclenchée :

```
$list.Add('Test')
Impossible de convertir l'argument «arg» (valeur «Test») de «Add» en type
«System.Int32»:...
```

Une classe générique peut avoir plusieurs paramètres :

System.Collections.ObjectModel.KeyedCollection<TKey, TItem>

Le principe ne change pas :

```
class FileList:
 System.Collections.ObjectModel.KeyedCollection[System.String, System.IO.FileInfo]
{
    [void] Add([System.IO.FileInfo] $item)
    {

([System.Collections.ObjectModel.KeyedCollection[System.String,System.IO.FileInfo]]$this).A
dd($Item)
    }


    [System.String] GetKeyForItem([System.IO.FileInfo] $item)
    { return $Item.FullName  }
}


$List=[FileList]::New()
$F=Dir C:\Windows -File
$List.Add($F[0])
$list[0]
$list[$F[0].FullName] #case sensitive
```

6.4 Interface

Avec cette notion on s'intéresse davantage à ce que sait faire une classe, à ces capacités, et non à ce qu'elle est, ses caractéristiques. Une interface permet également de manipuler des objets provenant de hiérarchies de classes différentes ayant une capacité commune, par exemple une connexion réseau pour un serveur ou un téléphone portable. Chacune sait se connecter à un réseau, on peut envisager une interface *IConnectable.*

Sous Powershell on ne peut pas déclarer une interface, seule leur implémentation est possible.

Nous avons vu dans le chapitre sur l'héritage la syntaxe de déclaration, prenons pour notre exemple la capacité, pour un objet, de créer un double de soi :

```
class Test : ICloneable
```

Voyons maintenant l'implémentation, une interface propose uniquement des signatures, le reste est à notre charge. Les membres de cette interface sont documentés sur MSDN, on peut aussi afficher leur signature ainsi :

```
$ofs="`r`n"
[System.ICloneable].GetMembers().Foreach({$_.ToString()})
System.Object Clone()
```

Cette interface propose une seule méthode, d'autres peuvent posséder des propriétés, des indexeurs et des événements. Notez que les interfaces possédant des événements ne pourront être implémentées sous Powershell.

Pour notre exemple nous n'effectuerons pas une copie profonde, mais une copie simple :

```
class Test : ICloneable{
  [string] $Description
  [int] $Compteur

  Test( [string] $Description, [int] $Compteur) {
    $this.Description =$Description
    $this.Compteur=$Compteur
  }

  [System.Object] Clone(){
     Return $this.MemberwiseClone()
  }
}
```

Notre classe possède deux propriétés et implémente le membre *Clone()* proposé par l'interface.

Si vous ne déclarez pas un des membres de l'interface, une exception est déclenchée :

```
Error during creationg of type "Test". Error message:
Method 'Clone' in type 'Test' from assembly 'powershell, Version=0.0.0.0,
Culture=neutral, PublicKeyToken=null' does not have an implementation.
```

La méthode MemberwiseClone() se charge du travail. Cette méthode est héritée de la classe **System.Object**, comme toutes les classes dérivent de cette classe, la nôtre peut la réutiliser.

Dans la documentation citée la signature de cette méthode indique un membre protégé :

```
protected Object MemberwiseClone()
```

Sous Powershell tous les membres sont publics, dans les langages compilés en revanche on trouve la notion d'accès aux membres. En C#, le mot clé ***protected*** signifie, rapidement, que la méthode *MemberwiseClone()* est disponible uniquement pour les développeurs, l'utilisateur ne peut pas accéder à la mécanique interne à partir d'une instance, mais uniquement à la méthode Clone() :

```
$o=[Test]::New('Test',10)
$b=$o.Clone()
$o,$b
Description Compteur
----------- --------
Test              10
Test              10
```

Note : les deux objets **$o** et **$b** sont distincts, mais de même contenu.

Le code suivant est donc correct :

```
$o -eq $b
False
```

Il compare les adresses des objets pas leur contenu, si vous désirez modifier ce comportement vous devez surcharger la méthode Equals.

Il n'est pas nécessaire de redéclarer la méthode *Clone()* dans les classes dérivées :

```
class Test2 : Test {
  [Datetime] $Date

  Test2( [Datetime] $Date, [string] $Description, [int] $Compteur):
   Base($Description, $Compteur) {
    $this.Date=$Date
  }
}
$o=[test2]::New('10/10/2010','Test2',20)
$b=$o.clone()
```

Comme cette classe contient une nouvelle propriété, il serait toutefois nécessaire de redéclarer la méthode *Equals*.

Note avancée : Pour les interfaces explicites voir cette fonction.

6.5 Indexeur

Un indexeur permet de considérer une classe comme un tableau en utilisant l'opérateur [].

La classe n'est pas en soi un tableau, l'indexeur accède directement à une collection de la classe.

Les indexeurs étant des méthodes, Jason Shirk propose de placer sur la classe l'attribut DefaultMember. Sa présence autorise, sous Powershell, de redéclarer les accesseurs de la classe :

```
[System.Reflection.DefaultMember("Item")]
class Test
{
    Hidden [int] get_Item([int]$i) { return $i * 2 }
}

$o=[Test]::new()
$o[2]
42
```

Ici '*Item'* est le nom par défaut de la propriété associé à l'indexeur, notez que seul les accesseurs sont implémentés, la propriété '*Item'* n'existe pas en tant que membre :

```
$o|gm -MemberType Property -force
#Aucun affichage
```

Dans l'exemple suivant basé sur un tableau d'entier, on redéfinie le nom de la propriété '*Tab*' afin de pointer sur une propriété existante de la classe :

```
 [System.Reflection.DefaultMember("Tab")]
class MyClass
{
    Hidden [int[]] $Tab = (new-object int[] 100)
    [int] get_Tab([int]$i) { return $this.Tab[$i]}
    [void] set_Tab([int]$i,[int] $Value) { $this.Tab[$i]=$Value}
}
$o=[MyClass]::New()
$o.Tab=0..99
$o[3]
3
```

La méthode *get_Tab* attend un entier, l'index, et renvoi un objet du même type que celui contenu dans le tableau $Tab.

La méthode *set_Tab* attend un entier, l'index, et un objet du même type que celui contenu dans le tableau $Tab. Comme c'est une affectation, elle ne renvoie pas de donnée.

On peut vouloir surcharger la méthode *get_Tab* afin de rechercher un élément soit par un numéro d'index soit par son nom :

```
[System.Reflection.DefaultMember("Tab")]
class MyClass
{
    Hidden [string[]] $Tab=(new-object string[] 10)
     #renvoie une chaîne
    [string] get_Tab([int]$i) { return $this.Tab[$i]}

     #renvoie un entier
    [Int] get_Tab ([string]$name) {
       return [array]::IndexOf($this.Tab,$name)
    }
    [void] set_Tab([int]$i,[string] $name){
       $this.Tab[$i]=$name
    }
}
$o=[MyClass]::new()
$o[0]='Test'
$o[1]='Indexer'
$o[0]
Test
$o['Indexer']
1
```

Attention dans cet exemple la recherche est sensible à la casse. La classe NameValueCollection propose également ce type d'accès.

6.5.1 Usage dans une boucle Foreach

Si on considère cette classe comme une collection on aimerait pouvoir la manipuler dans une boucle *Foreach*. L'implémentation précédente renvoi un seul objet, l'instance.

Pour implémenter le comportement du *Foreach* sur cette classe on doit hériter de l'interface ***IEnumerable***. Celle-ci impose l'implémentation de la méthode *GetEnumerator()* qui renvoie l'énumérateur de la propriété utilisée par l'indexeur. Ainsi l'instruction Foreach énumère le tableau de l'instance et non plus l'instance :

```
[System.Reflection.DefaultMember("Tab")]
class MyClass:System.Collections.IEnumerable
{
    Hidden [string[]] $Tab=(new-object string[] 10)
    [string] get_Tab([int]$i) { return $this.Tab[$i]}

    [int] get_Tab ([string]$name) {
```

```
        return [array]::IndexOf($this.Tab,$name)
    }
    [void] set_Tab([int]$i,[string] $name){
       $this.Tab[$i]=$name
    }
    [System.Collections.IEnumerator] GetEnumerator(){
        #On propage l'énumérateur
      return $this.Tab.GetEnumerator()
    }
}
$o=[myclass]::New()
$o[0]='zero' ; $o[1]='un' ; $o[2]='deux' ;$o[3]='trois'
$o|% {$_}
zero …
Foreach ($i in $o) {$i}
zero …
```

6.5.2 Redéfinir la propriété Count

Depuis la version 3 Powershell 'simule' la présence de deux propriétés, *Count* et *Length*. Dans notre cas si on les utilise elles renvoient 1. Si on considère cette classe comme une collection utilisable dans un foreach, il y a une décision à prendre. Soit on laisse en l'état soit on implémente une propriété *Count* renvoyant le nombre d'élément de la collection.

Une possibilité à l'aide d'une interface qui permet de déclarer une propriété et de redéfinir ses accesseurs :

```
Add-type @'
namespace Test {
  public interface ICountable
  {
      int Count
      {
          //Propriété en lecture seule
         get;
      }
  }
}
'@

@'
[System.Reflection.DefaultMember("Tab")]
class MyClass:System.Collections.IEnumerable,Test.ICountable
{
```

```
    Hidden [string[]] $Tab=(new-object string[] 10)
    [string] get_Tab([int]$i){
       Write-Warning 'Get tab Int';return $this.Tab[$i]
    }

    [int[]] get_Tab ([string[]]$Names) {
       Write-warning 'Get Array int[]'
       [int[]]$result=$Names|% {[array]::IndexOf($this.Tab,$_)}
       return $result
    }

    [int] get_Tab ([string]$name) {
       Write-warning 'Get int'
       return [array]::IndexOf($this.Tab,$name)  #Case sensitive
    }

    [void] set_Tab([int]$i,[string] $name){
       Write-Warning 'Set tab $Name'
       $this.Tab[$i]=$name
    }

    [System.Collections.IEnumerator] GetEnumerator(){
      Write-Warning 'GetEnumerator'
      return $this.Tab.GetEnumerator()
    }

  hidden [int] get_Count(){
    Write-warning "Get"
    return $this.Tab.Length
  }
}
'@ > C:\temp\MyClass.ps1
. C:\temp\MyClass.ps1

$o=[myclass]::New()
$o[0]='zero'
$o[1]='one'
$o[2]='two'
$o[3]='three'
$o
WARNING: GetEnumerator
```

```
Zero …
$o.Count
WARNING: Get
10
```

L'interface **ICountable** déclare sa propriété *Count* en lecture seule pour interdire sa modification, ici l'accesseur *get* peut être redéclaré sur une propriété :

```
$o.count=5
'count' is a ReadOnly property.
```

L'accès à la propriété *Length* déclenche l'itération automatique sur le tableau :

```
$o.Length
WARNING: GetEnumerator
4 …
```

Pour annuler ce comportement on peut créer un alias pointant sur la propriété *Count* :

```
Update-TypeData -TypeName myclass -MemberType AliasProperty -MemberName
Length -Value Count
$o.Length
WARNING: Get
10
```

Si un indexeur peut être considéré comme du sucre syntaxique, celui proposé par Powershell peut provoquer quelques collisions.

6.5.3 Accès multiple

Un indexeur peut récupérer plusieurs valeurs en un appel, modifions l'exemple précédent en y ajoutant la surcharge de méthode suivante :

```
    [int[]] get_Tab ([string[]]$names) {
       Write-warning 'Get Array int[]'
       [int[]]$result=$Names|% {$this.get_Tab($_)}
       return ,$result
    }
```

Rappel : la présence de la virgule précise l'émission d'un seul objet, un tableau.

Essayons :

```
$o['one','three']
WARNING: Get int
WARNING: Get int
1
3
```

Ce code fonctionne, mais pas de la manière attendue, ici on déclenche l'*array slicing*' qui renvoi les éléments demandés.

Pour éviter ce déclenchement prioritaire on doit utiliser le nom de la méthode au lieu de la syntaxe abrégée :

```
$o.get_Tab(@('one','three'))
WARNING: Get int
-1
```

Mais on doit typer le tableau afin de sélectionner la méthode adéquate :

```
$o.get_Tab([string[]]@('one','three'))
WARNING: Get Array int[]
1
3
```

Pour ce cas, on peut se contenter du comportement de Powershell.

6.5.4 Interface et indexeur

Une interface pouvant déclarer un indexeur :

```
Add-type @'
namespace Test {
  public interface IMyStringIndexeur
  {
      string this[int index]      // Indexer declaration
      {
          get;
          set;
      }
  }
}
'@
```

On utilisera le nom de propriété '*Item*' sur l'attribut DefaultMember :

```
@'
[System.Reflection.DefaultMember("Item")]
class MyClass : Test.IMyStringIndexeur
{
    Hidden [string[]] $Tab=(new-object string[] 10)

    [string] get_Item ([int]$index) {
       Write-warning 'get_Item'
       return $this.Tab[$index]
    }
    [void] set_Item([int]$index,[string] $name){
       Write-warning 'set_Item'
       $this.Tab[$index]=$name
```

```
    }
}
'@ > C:\temp\MyClass.ps1
. C:\temp\MyClass.ps1

$o=[myclass]::New()
$o[0]='zero'
$o[0]
zero
```

6.6 Délégué

Un délégué est similaire à un pointeur de fonction. Il peut contenir une ou plusieures références.

On ne peut pas créer de délégué, mais en utiliser. Depuis la version 4 leur gestion est facilitée :

```
class Test {
  [System.Predicate [System.Int32]] $Predicate= {param($i); $i -le 10}
}
$o=[Test]::New()
```

Ce délégué est du type Int, la collection sur laquelle l'appliquer doit être du même type :

```
[Int32[]]$T=1..10
[Int32[]]$T2=1..11
[System.Array]::TrueForAll($T,$O.Predicate)
true
[System.Array]::TrueForAll($T2,$O.Predicate)
false
```

L'exemple suivant applique un délégué sur un tableau :

```
class Test {
  [int[]] $Tab=1..10
  [System.Predicate [System.Int32]] $Predicate= {
    param($i); $i -le 10
  }
  [System.Func [System.Int32, System.Int32]] $Func={
    param($i); ($i % 2) -eq 0
  }
  [boolean] ExecutePredicate() {
     foreach ($i in $this.Tab) {
       if (-not $this.Predicate.Invoke($i))
       {return $false}
     }
     Return $true
  }
```

```
  [int[]] ExecuteFunc() {
     [int[]]$result=Foreach ($i in $this.Tab) {
       if ($this.Func.Invoke($i))
       {$i}
     }
     Return $result
  }
}
$o=[Test]::New()
```

Vous pouvez utiliser les types de délégués suivants : *Action, Converter, Func et Predicate*.

```
$o.ExecutePredicate()
True
$o.ExecuteFunc()
2 4 6 8 10
```

Le paramétrage de traitement s'en trouve facilité.

```
$o.Func={ param($i); ($i % 2) -ne 0  }
$o.ExecuteFunc()
1 3 5 7 9
```

6.7 Variable liée

Une variable liée permet d'exécuter du code lors de la lecture de son contenu et c'est l'exécution du code qui produit son contenu. Les classes permettent d'implémenter des variables liées :

```
class NowVariable : System.Management.Automation.PSVariable {
 NowVariable():base("Now", 0, "ReadOnly,AllScope") {}

 [object] get_Value(){ return  [System.DateTime]::Now }
 [string] ToString(){ return ($this.get_Value()).ToString()}
}
$Date=[NowVariable]::new()
$Date.Value
#ras
```

Comme l'accès à la propriété *$Date.Value* n'appelle pas la surcharge de l'accesseur, on doit appeler directement la méthode de l'accesseur *$Date.get_Value()* .

Ensuite il reste à déclarer une variable Powershell à partir de l'objet créé :

```
$ExecutionContext.SessionState.PSVariable.Set([NowVariable]::new())
$now
Tuesday, October 6, 2015 3:09:22 PM
"$now"
10/06/2015 15:09:25
```

6.8 Surcharge d'opérateur

Comme l'indique la documentation de Powershell :

«*Un opérateur est un élément de langage que vous pouvez utiliser dans une commande ou dans une expression.*»

Les plus connus étant les opérateurs arithmétiques (+, -, *, /, %) ou les opérateurs de comparaison (-eq, -ne, -gt, *etc*).

Pour une classe, la surcharge d'opérateur autorise la redéfinition de l'action d'un opérateur. Dans l'exemple suivant on ne peut pas additionner un objet de type entier et un objet de type Process :

```
[int]$i=10
$Process= Get-Process -id $pid
$i+$Process
```

```
Échec lors de l'appel de la méthode, car [System.Diagnostics.Process] ne
contient pas de méthode nommée «op_Addition».
```

Le message d'erreur reste à interpréter, il nous indique que la classe Process n'implémente pas l'opérateur d'addition (+) associé en interne à la méthode nommée ***op_Addition***.

On peut afficher la liste de ces méthodes d'opérateur implémentées :

```
[DateTime].GetMembers().Name
```

Et si le type ciblé surcharge un de ces opérateurs, on affichera sa signature ainsi :

```
[DateTime].GetMethod('op_Addition').ToString()
```

```
System.DateTime op_Addition(System.DateTime, System.TimeSpan)
```

Pour une nouvelle classe :

```
class Test
{
    [int] $Count

    Test([int] $i){
      Write-verbose "New Test(int)"
      $this.Count=$i
    }
}
```

L'addition de deux de ces instances provoque la même erreur :

```
$A=[Test]10
$B=[Test]20
$A+$B
```

```
Échec lors de l'appel de la méthode, car [Test] ne contient pas de méthode
nommée «op_Addition».
```

Bien évidement on peut se contenter de l'écriture suivante :

```
$A.Count + $B.Count
```

On peut aussi utiliser cette approche :

```
($A,$B|Measure-Object -Sum -Property Count).Sum
30
```

On détermine déjà sur quoi porte cette addition, ici il s'agit de la propriété *Count*.

Si on souhaite proposer une syntaxe plus concise, on peut implémenter l'opérateur d'addition :

```
class Test
{
    [int] $Count

    Test([int] $i){
      Write-verbose "New Test(int)"
      $this.Count=$i
    }

    static [Test] op_Addition([Test] $T1,[Test] $T2)
    {
        Write-verbose ("Call Op_Addition {0} {1}" -f $T1.Count,$T2.Count)
        return [Test]::New($T1.Count+$T2.Count)
    }
}
```

Comme il n'existe pas sous Powershell de mot clé implémentant la surcharge d'opérateur, on doit utiliser les noms de fonctions IL, ***op_Addition*** est un de ceux-là.

Testons notre nouveau code :

```
$verbosepreference='Continue'
$A=[Test]10
VERBOSE: New Test(int)
$B=[Test]20
VERBOSE: New Test(int)
$C=$A+$B
VERBOSE: Call Op_Addition 10 20
VERBOSE: New Test(int)
```

L'addition des deux instances renvoie une troisième instance du type [Test] créée dans la méthode ***op_Addition*** :

```
$C
Count
-----
   30
```

Sa propriété *Count* contient la valeur attendue. On peut enchainer plusieurs additions :

```
$A+$B+$C
VERBOSE: Call Op_Addition 10 20
VERBOSE: New Test(int)
VERBOSE: Call Op_Addition 30 30
VERBOSE: New Test(int)
Count
-----
   60
```

On peut également additionner une instance de la classe [Test] avec un entier :

```
10+$A #ou $A+10
VERBOSE: New Test(int)
VERBOSE: Call Op_Addition 10 10
VERBOSE: New Test(int)
Count
-----
   20
```

L'affichage nous indique deux créations d'objet, ceci est dû à la conversion implicite de la valeur 10 en une instance de la classe [Test] via le constructeur ***Test([int] $i)***.

C'est la signature de la méthode *op_Addition* qui déclenche cette conversion, car elle attend deux paramètres de type [Test] :

```
static [Test] op_Addition([Test] $T1,[Test] $T2)
```

Dans le cas où ce constructeur n'est pas déclaré, l'addition provoque l'erreur suivante :

```
Cannot convert argument "1", with value: "10", for "op_Addition" to type
"Test": "Cannot convert the "10" value of type "System.Int32" to type
"Test"."
```

Le code existant crée donc une instance transitoire afin de récupérer la valeur 10 !

Pourquoi faire simple quand on peut faire compliqué !

6.8.1 Additionner des choux ou des carottes

Que donne l'addition de choux et de carottes ? De la purée ? Une chote ou un caroux ? Une erreur ? Grande question !

Voyons de plus près le cas où le constructeur ***Test([int] $i)*** n'est pas déclaré :

```
class Test
{
    [int] $Count

    Test(){ Write-verbose "New Test()" }

    static [Test] op_Addition([Test] $T1,[Test] $T2)
    {
        Write-verbose ("Call Op_Addition {0} {1}" -f $T1.Count,$T2.Count)
        $result=[Test]::New()
        $result.Count=$T1.Count+$T2.Count
        return $result
    }
}
$A=[Test]@{Count=10}
$B=[Test]@{Count=10}
$A+$B
```

On utilise désormais une hashtable pour construire notre instance, l'addition de deux instances fonctionne toujours. Par contre l'addition d'une instance de la classe [Test] avec un entier n'est plus possible :

```
$A+10 #ou 10+$A
Cannot convert argument "1", with value: "10", for "op_Addition" to type
"Test": "Cannot convert the "10" value of type "System.Int32" to type
"Test"."
```

Si on souhaite autoriser ce cas, il nous faut déterminer le type du résultat. Est-ce [Test] ou [Int] ? A moins qu'il soit possible de choisir au cas par cas.

Pour le choix [Test], on peut déclarer deux autres méthodes *op_Addition* afin de modifier le type des opérandes :

```
#$A+10
    static [Test] op_Addition([Test] $T,[int] $i)
    {
        Write-verbose ("Call [Test] Op_Addition([Test],[int]) {0} {1}" -f $T.Count,$i)
        $result=[Test]::New()
        $result.Count=$T1.Count+$i
        return $result
```

```
    }
#10+$A
    static [Test] op_Addition([int] $i,[Test] $T)
    {
        Write-verbose ("Call [Test] Op_Addition([Test],[int]) {0} {1}" -f $i,$T.Count)
        $result=[Test]::New()
        $result.Count=$T1.Count+$i
        return $result
    }
```

Le type du résultat est toujours une instance de la classe [Test]. Le code étant identique une factorisation est souhaitable.

Pour le choix [int], ces déclarations de surcharge de méthode ne seront pas autorisées, car le type de la valeur de retour n'est pas pris en considération lors de la différenciation des surcharges :

```
    static [int] op_Addition([Test] $T,[int] $i)
    static [int] op_Addition([int] $i,[Test] $T)
```

6.8.2 Un peu de cast..agne

Pour notre addition **$A+10**, nous savons ceci (un peu d'évidence ne nuit à personne ☺) :

```
$A -is [Int]
False
10 -is [Test]
False
```

Puisqu'on ne peut pas utiliser une autre surcharge de l'opérateur d'addition, essayons de transformer une des opérandes :

```
($A -as [int])+10
10
```

Cette transformation renvoi $null, qui est converti en 0, les règles de conversions de Powershell s'appliquent, comme indiqué dans les spécifications de Powershell, chapitre 6.4 *Conversion to integer*, où on y lit également ceci :

« For other reference type values, if the reference type supports such a conversion, that conversion is used; otherwise, the conversion is in error. »

Le code suivant correspond au dernier cas, il n'existe pas de code de conversion :

```
[int]$i=$A
Cannot convert the "Test" value of type "Test" to type "System.Int32".
[Test]$C=10
Cannot convert the "10" value of type "System.Int32" to type "Test".
```

Pour notre addition nous allons surcharger l'opérateur de conversion implicite. Le nom de sa méthode est ***op_Implicit***, il attend un paramètre de type [Test] et renvoi une valeur de type [int] :

```
class Test
{
    [int] $Count
    Test(){ Write-verbose "New Test()" }
    static [Test] op_Addition([Test] $T1,[Test] $T2)
    {
        Write-verbose ("Call Op_Addition {0} {1}" -f $T1.Count,$T2.Count)
        $result=[Test]::New()
        $result.Count=$T1.Count+$T2.Count
        return $result
    }

     static [int] op_Implicit([Test]$T)
     {
         Write-verbose "Call cast Implicit . Return int"
         return $T.Count
     }
}
$A=[Test]@{Count=10};$B=[Test]@{Count=10}
$A+$B
Count
-----
   20
10+$A
VERBOSE: Call cast Implicit . Return int
20
```

L'addition est effectuée sans créer une nouvelle instance, le résultat est du type [int]. Vérifions maintenant l'inversion des opérandes :

```
$A+10
Cannot convert argument "1", with value: "10", for "op_Addition" to type
"Test": "Cannot convert the "10" value of type "System.Int32" to type
"Test"."
```

Cela ne fonctionne pas, car dans les spécifications de Powershell, chapitre *7.7.1 Addition*, il est indiqué :

« The result of the addition operator + is the sum of the values designated by the two operands after the usual arithmetic conversions (§6.15) have been applied.

***This operator is left associative**.»*

L'opérateur d'addition est associatif à gauche, ce qui veut dire que les opérations sont effectuées de gauche à droite. C'est donc l'opérande gauche qui détermine le type ciblé lors de la conversion.

On doit ajouter une surcharge de la méthode ***op_Implicit*** avec un paramètre de type [Int] et une valeur de retour de type [Test] (l'inverse de la surcharge précédente) :

```
    static [Test] op_Implicit([int]$I) {
        Write-verbose "Call cast Implicit . Return Test"
        $result=[Test]::New()
        $result.Count=$I
        return $result
    }
```

Cela fonctionne, mais on revient au point de départ, car il y a deux créations d'instance :

```
$A+10
VERBOSE: Call cast Implicit . Return Test
VERBOSE: New Test()
VERBOSE: Call Op_Addition TEST 10 10
VERBOSE: New Test()
Count
-----
  20
```

Le code final pouvant être celui-ci :

```
class Test {
    [int] $Count

    Test(){ Write-verbose "New Test()" }

#    Test([int] $i){
#       Write-verbose "New Test(int)"
#       $this.Count=$i
#     }

    static hidden [Test] Init([int] $Total) {
        Write-verbose "New Init()"
        $result=[Test]::New()
        $result.Count=$Total
        return $result
    }

    static [Test] op_Addition([Test] $T1,[Test] $T2)
    {
        Write-verbose ("Call [Test] Op_Addition([Test],[Test]) {0} {1}" -f
$T1.Count,$T2.Count)
        return ([Test]::Init($T1.Count+$T2.Count))
    }
```

```
    static [Test] op_Addition([Test] $T,[int] $i)
    {
        Write-verbose ("Call [Test] Op_Addition([Test],[int]) {0} {1}" -f
$T.Count,$i)
        return ([Test]::Init($T.Count+$i))
    }

    static [Test] op_Addition([int] $i,[Test] $T)
    {
        Write-verbose ("Call [Test] Op_Addition([int],[Test]) {0} {1}" -f
$i,$T.Count)
        return ([Test]::Init($T.Count+$i))
    }

    static [Test] op_Implicit([int]$I)
    {
        Write-verbose "Call cast Implicit to [Test]"
        $result=[Test]::New()
        $result.Count=$I
        return $result
    }
    static [int] op_Explicit([Test]$T)
    {
        Write-verbose "Call cast Explicit to [int]"
        return $T.Count
    }
}
$VerbosePreference='Continue'
$i=10
[Test]$A=[Test]@{Count=123}
```

Si on ne type pas la variable $A les opérateurs de conversion ne seront pas mis à profit puisqu'une variable sans type peut se voir affecter n'importe quel contenu, ici pas besoin de conversion.

Pour la règle suivante :

- ***Cast conversion.*** *If the target type defines a implicit or explicit cast operator from the source type, use that. If the source type defines an implicit or explicit cast operator to the target type, use that.*

Elle concerne la création d'instance, ce qui fait que le comportement des conversions explicite et implicite ne se calque pas tout à fait sur l'écriture utilisé en C#, mais le résultat est identique :

```
$x=[int]$A
COMMENTAIRES : Call cast Explicit to [int]
$A=[Test]$i
COMMENTAIRES : Call cast Implicit to [Test]
COMMENTAIRES : New Test()
$i -as [Test]
COMMENTAIRES : Call cast Implicit to [Test]
COMMENTAIRES : New Test()
Count
-----
   10
$A -as [Int]
COMMENTAIRES : Call cast Explicit to [int]
10
```

Il est recommandé d'utiliser l'opérateur –*as* et de vérifier le résultat de la conversion.

De plus pour le transtypage, si Powershell ne peut pas passer par la porte, il essaie de passer par la fenêtre :

- ***IConvertible conversion.*** *If the* source type *defines an [IConvertible] implementation that knows how to convert to the target type, use that.*

Attention donc aux conversions implicites de Powershell, voir à la précédence des opérateurs :

```
10.1+$a
VERBOSE: Call [Test] Op_Addition([int],[Test]) 10 123
[double]$D=10+$a
VERBOSE: Call cast Implicit to [int]
[double]$D=10.1+$a
VERBOSE: Call [Test] Op_Addition([int],[Test]) 10 123
VERBOSE: New Init()
VERBOSE: New Test()
Cannot convert the "Test" value of type "Test" to type "System.Double".
```

Le nombre 10.1 est du type double, aucune surcharge d'opérateur ne traite ce type, en revanche dans la liste des opérateurs la valeur 10.1 peut être converti en un entier et ainsi matcher une des surcharges :

```
[Test].GetMethods()|? name -eq 'op_Addition'|% {$_.tostring()}
Test op_Addition(Test, Test)
Test op_Addition(Test, Int32)
Test op_Addition(Int32, Test)  <---
```

Il reste aussi ce cas :

```
Write-host (10+$a)
133
Write-host (10.1+$a)
Test
```

La surcharge de la méthode *ToString()* reste à implémenter.

Allez, une dernière pour la route, le résultat de Trace-Command devient verbeux :

```
Trace-Command TypeConversion -ex {[double]$D=10.1+$a} -pshost
DEBUG: TypeConversion Information: 0 : Converting "System.Object[]" to "System.Object". …
```

Ce qui est normal puisque le code des méthodes d'opérateur est du code Powershell qui alimente la même source ;-)

Donc, on peut surcharger les opérateurs, mais vous savez maintenant où vous mettez les pieds ☺

6.8.3 Liste des opérateurs

La page suivante, dédiée au C#, propose cette liste :

Opérateur	Usage	Nom (1) unaire (2) binaire	Nom de méthode alternative
-band	$x = $y -band $z	**op_BitwiseAnd** (2)	**BitwiseAnd**
-bor	$x = $y -bor $z	**op_BitwiseOr** (2)	**BitwiseOr**
-xor	$x = $y -xor $z	**op_ExclusiveOr** (2)	**Xor**
-eq	If ($x -eq $y)	**op_Equality** (2)	**Equals**
-gt	If ($x -gt $y)	**op_GreaterThan** (2)	**Compare**
-ge	If ($x -ge $y)	**op_GreaterThanOrEqual** (2)	**Compare**
-ne	If ($x -ne $y)	**op_Inequality** (2)	**Compare**
-lt	If ($x -lt $y)	**op_LessThan** (2)	**Compare**
-le	If ($x -le $y)	**op_LessThanOrEqual** (2)	**Compare**
-not	If (-not $x)	**op_LogicalNot** (1)	**Not**
+	$x = $y + $z	**op_Addition** (2)	**Add**
++	$x++	**op_Increment** (1)	**Increment**
%	$x = $y % $z	**op_Modulus** (2)	**Modulus**
*	$x = $y * $z	**op_Multiply** (2)	**Multiply**
/	$x = $y / $z	**op_Division** (2)	**Divide**
-	$x = $y – $z	**op_Subtraction** (2)	**Subtract**
--	$x--	**op_Decrement** (1)	**Decrement**
-shr	$x = $y -shr 2	**op_RightShift** (2)	**RightShift**
-shl	$x = $y -shl 2	**op_LeftShift** (2)	**LeftShift**
- (unary)	$x = -$test1Object	**op_UnaryNegation** (1)	**Negate**
+ (unary)	[int] $x = +$test1Object	**op_UnaryPlus** (1)	**Plus**
Explicit	[String] $x = [string]$test1Object	**op_Explicit** (1)	**ToXxx** or **FromXxx**
Implicit	[String] $x = $test1Object	**op_Implicit** (1)	**ToXxx** or **FromXxx**
-bnot	$x = -bnot $x	**op_OnesComplement** (1)	**OnesComplement**

Une surcharge de l'opérateur *, par exemple, est également une surcharge de l'opérateur ***=**.
Powershell ne contrôle pas l'appairage des opérateurs, par exemple si on redéfini l'opérateur **–eq** on doit également le faire pour l'opérateur **–ne.**

Il est recommandé d'implémenter l'interface IComparable si on redéfini les opérateurs **–lt** et **–gt.**

6.9 Sérialisation

L'usage d'une instance de classe dans un job ou un traitement distant est possible, elle sera sérialisé comme tout autre objet et sera désérialisé en une instance du type PSObject.

On peut vouloir le réhydrater, c'est à dire la reconstruire dans le même type d'origine, dans ce cas, bien évidemment, les contraintes sont les suivantes :

1. la classe doit exister des deux côtés (Local/Distants),
2. sa définition doit être identique.

Pour une classe comportant des propriétés de type scalaire l'opération est facilitée par cette possibilité native :

> ***PSObject property conversion.*** *If the source type is a PSObject, try to create an instance of the destination type using its default constructor, and then use the property names and values in the PSObject to set properties on the source object. If a name maps to a method instead of a property, invoke that method with the value as its argument.*

Le code suivant utilise cette possibilité :

```
@'
class Computer {
     [string] $Nom;
     [string] $OS;

    Computer(){}

    Computer([string] $Nom, [string] $OS)
    {
      $this.Nom = $Nom
      $this.OS = $OS
    }
}
'@ > c:\temp\ComputerClass.ps1
. c:\temp\ComputerClass.ps1

$Object=Start-job {
  . c:\temp\ComputerClass.ps1
  [Computer]::New('Test', 'Windows10')
} |
 Wait-Job|
 Receive-job -AutoRemoveJob -Wait
```

Une fois reçu l'objet crée dans le job :

```
$object.GetType().FullName
System.Management.Automation.PSObject
$object
RunspaceId : 131a2001-951a-4838-bd2a-3bad10fcc1a1
Nom        : Test
OS         : Windows10
```

On peut le transtyper dans sa classe d'origine :

```
$computer=[Computer]$object
$computer.gettype().fullname
Computer
$computer
Nom  OS
---  --
Test Windows10
```

Si le constructeur par défaut n'existe pas l'erreur suivante est déclenchée :

```
$Server=[Computer]$object
Impossible de convertir la valeur « Computer » du type
« Deserialized.Computer » en type « Computer ».
```

L'imbrication d'objet fonctionne tant que chaque classe Powershell déclare un constructeur par défaut :

```
@'
class Computer {
    [string] $Nom
    [string] $OS
    [Computer] $DC
    Computer(){}
    Computer([string] $Nom, [string] $OS) {
      $this.Nom = $Nom
      $this.OS = $OS
      $this.DC=$Null
    }
    Computer([string] $Nom, [string] $OS,[string] $DCName) {
      $this.Nom = $Nom
      $this.OS = $OS
      $this.DC=[Computer]::New('PrimaryDC','Windows 2012 R2')
    }
}
'@ > c:\temp\ComputerClass.ps1
. c:\temp\ComputerClass.ps1
```

Il est toutefois nécessaire de préciser la profondeur de la sérialisation :

```
$Object=Start-job {
  . c:\temp\ComputerClass.ps1
  Update-TypeData -TypeName Computer -SerializationDepth 2 -Force
  [Computer]::New('Test', 'Windows10')
  [Computer]::New('Test', 'Windows10','DC001')
} |
 Wait-Job|
 Receive-job -AutoRemoveJob -Wait
 #Réhydratation
$Computer=[Computer]$object[0]
$Computer1=[Computer]$object[1]
$Computer.GetType().fullname
Computer
$Computer1.DC.GetType().fullname
Computer
```

Depuis la version 3 il est possible d'utiliser la classe PSSerializer :

```
$Serial=[System.Management.Automation.PSSerializer]::Serialize($Computer)
$Object=[System.Management.Automation.PSSerializer]::Deserialize($Serial)
```

Plus besoin d'utiliser un disque physique comme avec le cmdlet Export-CliXml.

Enfin on peut gérer le versionning des classes hébergées dans des modules en côte à côte en ajoutant les informations nécessaires :

```
public interface IVersionable
{
   Microsoft.PowerShell.Commands.ModuleSpecification ModuleSpecification
   {
      get;
      set;
    }
}
```

La classe ***ModuleSpecification*** est réhydratée par Powershell. En revanche on doit préciser sa propriété *RequiredVersion.* A suivre…

7 Interaction C# et classe PowerShell

Voyons quelques cas d'usage de classes Powershell avec le C#.

7.1 Add-Type

Le scénario suivant crée une classe via Add-Type

```
Add-type -TypeDefinition @'
  public class Test
  {
      public string Name;
  }
'@

$o=[Test]::new()
$o|Get-Member
```

```
   TypeName: Test
Name        MemberType   Definition
----        ----------   ----------
...
Name        Property     string Name {get;set;}
```

Puis une classe Powershell de même nom :

```
class Test
{
    [string] $SurName
    [int] $Age
}
```

L'exécution de cette déclaration ne déclenche pas de collision de nom, il n'y a donc ni erreur ni warning :

```
$o=[Test]::new()
$o|Get-Member
```

```
...
Age         Property     int Age {get;set;}
SurName     Property     string SurName {get;set;}
```

Mais la classe [Test] créée par Add-Type se retrouve masquée.

La classe *[Test]* créée par ***Add-Type*** existe toujours :

```
Add-type -TypeDefinition @'
   public class Test
   {
       public string MyName;
   }
'@
Add-type : Cannot add type. The type name 'Test' already exists.
```

On ne peut pas l'écraser par une redéfinition, à la différence d'une classe Powershell :

```
class Test {  [Int] $Count }
$o=[Test]::new()
$o|Get-Member
...
Count       Property     int Count {get;set;}
```

Dans une nouvelle session, si on inverse l'ordre de création de classe de même nom :

```
class Test {
    [string] $SurName
    [int] $Age
}
$o=[Test]::new()
$o|Get-Member
…
Age         Property     int Age {get;set;}
SurName     Property     string SurName {get;set;}
Add-type -TypeDefinition @'
   public class Test
   {
       public string Name;
   }
'@
```

La classe *[Test]* manipulée reste la classe Powershell, celles-ci priment donc sur les classes de même nom créée par ***Add-Type*** :

```
$o=[Test]::new()
$o|Get-Member
…
Age         Property     int Age {get;set;}
SurName     Property     string SurName {get;set;}
```

La classe n'est pas écrasée :

```
Add-type -TypeDefinition @'
   public class Test
   {
       public string MyName;
   }
'@
Add-type : Cannot add type. The type name 'Test' already exists.
```

Ce n'est pas un bug, mais la conception des classes Powershell. Ce comportement permet de redéfinir un assembly et de décharger 'le précédent', sous réserve de libérer tous les objets associés à l'assembly.

Puisque l'on peut redéfinir une classe il faut bien évidement créer une nouvelle instance afin de bénéficier de la nouvelle version, les précédentes instances créées avant sa redéfinition restent valides.

Notez que pour différencier ces classes, on doit ajouter un espace de nom :

```
Add-type -TypeDefinition @'
Namespace Laurent {
   public class Test
   {
       public string MyName;
   }
}
'@
$o=[Laurent.Test]::new()
$o|Get-Member
   TypeName: Laurent.Test
…
Name        Property     string Name {get;set;}
```

Un espace de nom sert justement à éviter les collisions de noms de classe.

7.2 Héritage d'une classe C#

Le code suivant ne fonctionnera pas :

```
Add-type -TypeDefinition @'
  public class Parent
  {
      public string Name;
  }
'@

class Child:Parent {
  [string] $SurName
}
+ class Child:Parent {
Unable to find type [Parent].
[Parent]
Unable to find type [parent].
```

De par la conception des classes Powershell, la classe *[Parent]* doit être préalablement chargée en mémoire avant de parser la déclaration de classe *[Child]*.

Ceci fonctionne puisqu'on retarde l'analyse de la classe Powershell :

```
Add-type -TypeDefinition @'
  public class Parent
  {
      public string Name;
      protected int age;
  }
'@

@'
class Child:Parent {
  [string] $SurName
}
'@  > c:\temp\child.ps1
. c:\temp\Child.ps1

$o=[child]::new()
$o
SurName Name
------- ----
```

7.3 Héritage de classe entre modules

Pour le moment le scénario suivant est impossible :

```
@'
$script:Ver='Version module A'
  Class A {
    [string] Getver(){
      Write-warning "Classe A"
      Return $script:Ver
    }

    [string] GetName(){
      Write-warning "GetName. Classe A"
      Return $this.Getver()
    }
  }
  function getTypeA{[A]}
'@ > C:\temp\A_Class.psm1

@'
$script:Ver='Version module B'
  Class B:A {
   [string] GetName(){
     Write-warning "GetName. Classe B"
     Return $this.Getver()
   }
  }
  function getTypeB{[B]}
'@ > C:\temp\B_Class.psm1

ipmo C:\temp\A_Class.psm1
ipmo C:\temp\B_Class.psm1
Type [A] not found.
```

Le type du module A devant être accessible au module B lors de l'import, la solution est soit de compiler une classe puis de la charger, soit de déclarer temporairement un raccourci de type :

```
$samT='System.Management.Automation.TypeAccelerators'
$AcceleratorsType= [PSObject].Assembly.GetType($samT)
ipmo C:\temp\A_Class.psm1
```

```
$AType=getTypeA
$AcceleratorsType::Add('A',$AType) # Même nom de type
$m=ipmo C:\temp\B_Class.psm1 -PassThru
```

Une fois la liaison de type effectué on peut supprimer le raccourci :

```
$AcceleratorsType::Remove('A')
$b=getTypeB
$o=$b::new()
```

Le code d'une méthode de la classe de base, ici [A], référence son module, le module A.

Si une de ses méthodes est redéfinie dans la classe dérivée, ici [B], son code référencera le module B.

Il est donc possible qu'une méthode d'instance utilise plusieurs contextes de module :

```
$o.GetName()
WARNING: GetName. Classe B
WARNING: Classe A
Version module A
$o2=$AType::new()
$o2.GetName()
WARNING: GetName. Classe A
WARNING: Classe A
Version module A
```

8 DSC

L'ajout des trois attributs suivants autorisent la création de ressource DSC à l'aide de classes Powershell (classe basée ressource) : DscResource, DscProperty et DscLocalConfigurationManager.

DscLocalConfigurationManager s'applique à une configuration, cet attribut ne propose aucune propriété. Ce type de configuration est dédié aux managers DSC locaux.

DscResource s'applique à une classe, cet attribut ne propose aucune propriété. Sa présence indique que la classe est une ressource DSC.

DscProperty s'applique à une propriété, cet attribut propose les propriétés suivantes :

- *Mandatory* : La propriété est requise.
- *Key* : La propriété est une clé unique identifiant une ressource. Elle est donc implicitement requise (*mandatory*).
- *NotConfigurable* : La propriété n'est pas configurable. Son contenu est renseigné par la méthode Get() associée.

Une classe 'DSC' doit définir les trois méthodes **Set, Get** et **Test** ainsi qu'une propriété clé et un constructeur par défaut.

Une classe DSC doit être déclarée dans un module et celui-ci doit posséder un manifeste déclarant une nouvelle clé nommée '***DscResourcesToExpor***t'. Cette clé est un tableau de chaîne de caractères contenant les noms des classes basées ressources du module, elle facilite la découverte des noms de ressources. Voir également ce bug.

Ces évolutions permettent de déclarer plusieurs ressources dans un module.

9 Conclusion

Notez que l'AST a été enrichi pour traiter ces nouveaux mots clé, faute de temps je n'ai pas abordé ce sujet ni celui des traces d'exécution au sein des méthodes bien que la version 5 propose des cmdlets autour de ETW.

La possibilité de mixer du code Powershell et des classes facilite l'écriture et évite l'apprentissage d'un nouveau langage, il s'agit bien de classe dot Net, mais c'est un wrapper construit autour d'un assembly dynamique.

Pas de doute, la mécanique interne est remarquable, mais pour moi cela reste une adaptation des principes de la POO sous Powershell, cela me fait penser à une classe du type :

PowerShell : IJob, IEvent, IClass

C'est bien de la POO, mais elle a peu à voir avec les capacités de langages moderne tels que C#, Python, Delphi, etc. Ce qui engendre une certaine frustration.

L'impossibilité de créer des membres privés et des accesseurs est regrettable, tous les membres sont public, le mot clé *Hidden* masque un membre, mais ne le protège pas.

Pour le moment, comme nous avons pu le voir autour de l'exemple sur l'indexeur, il faut parfois combiner du C# et du PS tout en s'appuyant sur ETS pour annuler des comportements de Powershell !

Une amélioration serait appréciée ☺

Rien de bien compliqué pour quelqu'un connaissant les principes de la POO, sous réserve de connaitre certains de ses comportements, en revanche pour qui ne les connait cela peut s'avérer ardu de les aborder avec Powershell. Par exemple comment expliquer le concept d'interface sans pouvoir en créer ? Celui de propriété sans possibilité d'implémenter des accesseurs ? Ou encore celui de modificateurs d'accès quand tout est public ?

Ceci dit, l'apprentissage des classes pour un usage simple, comme vue dans le premier chapitre, n'est pas en soi insurmontable.

Ce document devra être corrigé lors de la disponibilité de la version RTM de PowerShell V5.