La création de membres d'objet synthétiques sous PowerShell

Par Laurent Dardenne, le 07 septembre 2008.

Niveau

Débutant	Avancé	Confimé

Le langage PowerShell est souvent perçu comme un langage orienté objet alors qu'il est un langage de shell basé objet, certes il autorise la création et la manipulation d'objets mais sa grammaire ne propose aucune possibilité de création de classe *ex-nihilo* comme le permet un langage orienté objet.

Il existe bien le cmdlet **Create-Object** mais il ne fait que créer une instance d'une classe déclarée au sein d'un assembly compilé.

Le système d'extension de type (ETS) permet soit d'étendre un type manipulé sous PowerShell à l'aide d'un fichier d'extension de type au format XML (.ps1xml), soit d'étendre un objet à la volée par l'ajout de membres « synthétiques » à l'aide du cmdlet **Add-member**. Ce tutoriel aborde uniquement cette seconde possibilité.

Chapitres

Pré-requis

Connaissance des principes et concepts de script de PowerShell ainsi que ceux de la POO.

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

Je tiens à remercier Vow et shawn12 pour leurs corrections orthographiques.

1 A propos du système d'extension de type

Afin d'uniformiser la représentation des objets, PowerShell utilise un système d'adaptation de type, voyons ce qu'il en est dit dans le SDK :

« ETS (système d'extension de type) résout deux questions fondamentales :

D'abord, certain objets .NET n'ont pas par défaut le comportement nécessaire pour agir en tant que données entre les cmdlets.

- *Certains objets .NET sont des «méta- objets» (par exemple : les objets de WMI, d'ADO ou d'XML) pour lesquels les membres décrivent les données qu'ils contiennent. Cependant, dans un environnement de scripting ce sont les données contenues qui sont les plus intéressantes, pas la description des données contenues. ETS résout cette question en présentant la notion d'adaptateur qui adapte un objet .NET sous-jacent afin d'avoir par défaut la sémantique attendue.*

- *Certains membres d'objet .NET sont nommés inconsidérément, ou bien fournissent un ensemble insuffisant de membres publiques, ou encore fournissent des possibilités insuffisantes. ETS résout cette question en introduisant la capacité d'étendre un objet .NET avec des membres additionnels.*

En second lieu, le langage de script de Windows PowerShell est sans type (typeless) car une variable n'a pas besoin d'être déclarée d'un type particulier. C'est-à-dire, les variables qu'un développeur de script crée sont par nature sans type (typeless). Cependant, le système de Windows PowerShell est piloté par les types (type-driven) parce qu'il dépend d'un nom de type pour fonctionner par exemple pour des opérations de base telles que l'affichage ou le tri.

Par conséquent un développeur de script doit être capable d'énoncer le type d'une de ses variables et de constituer son propre ensemble d'objets typés dynamiquement qui contiennent des propriétés et des méthodes et peuvent participer au système piloté par les types. ETS résout ce problème en fournissant un objet commun pour le langage de script qui a la capacité d'énoncer son type dynamiquement et d'ajouter des membres dynamiquement.

Fondamentalement, ETS résout la question mentionné précédemment en fournissant l'objet PSObject qui agit en tant que base d'accès de tous les objets du langage de script et fournit une abstraction standard pour le développeur de cmdlet.

…

Pour les développeurs de script, ETS fournit le support suivant :

- *La capacité de référencer tout type fondamental d'objet en utilisant la même syntaxe ($a.x).*

- *La capacité d'accéder au delà de l'abstraction fournie par l'objet PSObject (tel qu'accéder uniquement aux membres adaptés, ou d'accéder directement à l'objet de base).*
- *La capacité de définir les membres bien connus qui contrôlent le formatage, le tri, la sérialisation, et d'autres manipulations sur une instance ou un type d'objet.*
- *Les moyens d'appeler un objet comme d'un type spécifique et de contrôler ainsi l'héritage des membres du type de base.*
- *La capacité d'ajouter, d'enlever, et de modifier les membres étendus.*
- *La capacité de manœuvrer l'objet PSObject lui-même s'il y a lieu.»*

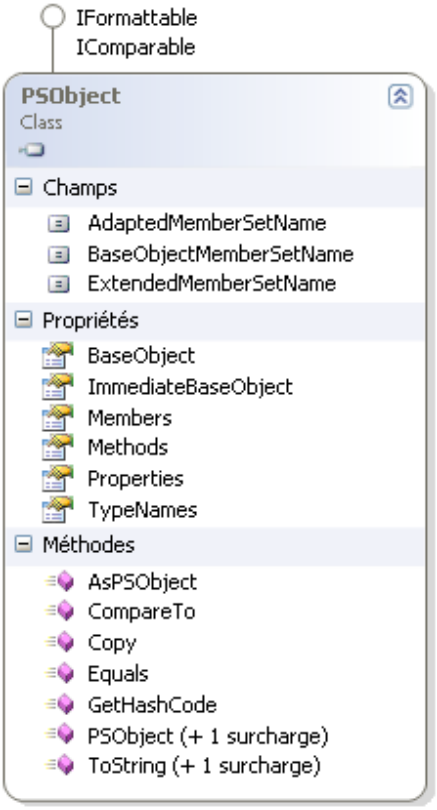
A propos des langages basé-objet :

http://en.wikipedia.org/wiki/Object-based_language

2 PSObjet

Comme nous le précise le SDK la classe **PSObject** sert de couche d'adaptation, elle est en quelque sorte un intermédiaire entre l'objet original et le shell.

La classe **PSObject** encapsule un objet de type **Object** ou **PSCustomObject :**



objet accessible par les propriétés *BaseObject* et *ImmediateBaseObjec*t, ces membres, étendues ou pas, étant accessibles par les autres propriétés qui sont du type **PSMemberInfoCollection<T>,** à l'exception de la propriété *TypeNames*.

Elle offre une vue cohérente de n'importe quel objet dans l'environnement de Windows PowerShell et sert de base à l'accès de tous les objets.

La création d'un tel objet se fait soit directement par l'appel au cmdlet **New-Object** en précisant la classe *System.Management.Automation.PsObject.*

Créons trois objets portant la même donnée, ici la date du jour :

```
$D=get-date
$pso = new-object System.Management.Automation.PsObject (Get-Date)
$pso2 = new-object System.Management.Automation.PsObject $D
$psc = new-object System.Management.Automation.PsObject
```

Affichons le type de chaque objet précédemment créé :

```
$d.gettype()
```

```
IsPublic  IsSerial   Name            BaseType
--------  --------   ----            --------
True      True       DateTime        System.ValueType
```

```
$pso.gettype()
```

```
IsPublic   IsSerial  Name            BaseType
--------   --------  ----            --------
True       True      DateTime        System.ValueType
```

```
$pso2.gettype()
```

```
IsPublic   IsSerial  Name            BaseType
--------   --------  ----            --------
True       True      DateTime        System.ValueType
```

Les trois premières approches sont identiques et l'affichage des membres des objets *$d*, *$pso* et *$pso2* renvoie les mêmes informations.

En revanche le dernier objet *$psc* est du type **PSCustomObject :**

```
$psc.GetType()
```

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

L'absence de paramètre crée un objet de type **PSCustomObject**, notez que cette classe ne possède pas de constructeur :

```
$pso = new-object System.Management.Automation.PSCustomObject
```

```
New-Object : Constructor not found. Cannot find an appropriate constructor for type
System.Management.Automation.PSCustomObject.
```

Et dans ce cas les propriétés *BaseObject* et *ImmediateBaseObjec*t ne sont pas renseignées.

Il est également possible de créer un objet en utilisant le cmdlet **Select-Object :**

```
$o = 1|Select-object Nom
```

Le cmdlet **Select-Object** crée, à partir d'un objet .NET, un objet de type **PSCustomObject** mais sachez que dans ce cas les tous les membres autres que ceux spécifiés ne seront plus accessibles. Bien que le type de l'objet passé dans le pipe, ici 1 de type integer, soit mémorisé :

```
$o|gm
```

```
   TypeName: Selected.System.Int32
…
```

```
$o.pstypenames
```

```
Selected.System.Int32
System.Management.Automation.PSCustomObject
System.Object
```

Note :

Les objets .NET sont adaptés le plus tard possible ce qui fait que dans certains cas ils ne se comportent pas comme des objets adaptés, pour plus de détails voir ce post de Bruce Payette (http://www.manning-sandbox.com/message.jspa?messageID=62388)

```
 #Objet natif
$D=[datetime]::now
$D
$D -is [PSObject]
```

```
False
```

```
$D.Hour>$null
 #L'objet natif est adapté lors de l'accès à une de ses propriétés
$D -is [PSObject]
```

```
True
```

Voir aussi :

Le pattern Adaptateur (de « Design patterns par la Pratique »)

http://www.editions-organisation.com/Chapitres/9782212111392/Shalloway_Chap_7.pdf?xd=de0af2602fe4d8adc5f209039fd89033

2.1 Membre de base, adapté, et étendu

Texte issu du SDK

"Conceptuellement, ETS emploie les termes suivants pour souligner le rapport entre les membres d'origine de l'objet de base et les membres supplémentaires ajoutés par Windows PowerShell.

2.1.1 Les membres de l'objet de base

*Si l'objet de base est indiqué en construisant un objet **PSObject**, alors les membres de l'objet de base sont rendus disponibles par la propriété **Members**.*

2.1.2 Les membres adaptés

Quand un objet de base est un méta-objet, qui contient des données de manière générique dont les propriétés « décrivent » les données contenues, ETS adapte ces objets dans une vue afin de

*permettre l'accès direct aux données par des membres adaptés de l'objet **PSObject**. Les membres adaptés et les membres de base de l'objet sont accessibles par la propriété **Members**.*

2.1.3 Membres étendus

*En plus des membres rendus disponibles à partir de l'objet de base ou des membres adaptés créés par Windows PowerShell, **PSObject** peut également définir des membres étendus qui étendent (enrichissent) l'objet de base original avec des informations additionnelles utiles dans l'environnement de scripting.*

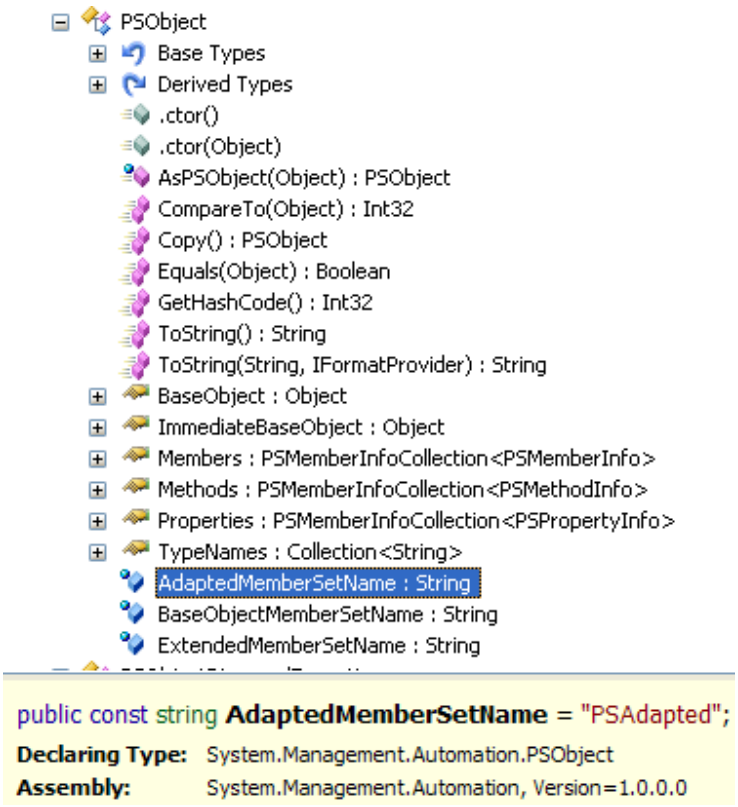
*Par exemple, tous les cmdlets natifs fournis par Windows PowerShell, tel que les cmdlets **Get-Content** et **Set-Content**, prennent un paramètre Path. Pour s'assurer que ces cmdlets, et d'autres, peuvent fonctionner avec des objets de différents types, un membre Path peut être ajouté aux objets de sorte qu'ils énoncent tous leur information d'une manière identique. Ce membre Path étendu fait en sorte que les cmdlets peuvent fonctionner avec ces différents types bien que la classe de base ne possède par de membre Path.*

*Les membres étendus, les membres adaptés, et les membres de l'objet de base sont accessible par la propriété **Members**."*

Comme indiqué dans le blog de l'équipe de développement de PowerShell (http://blogs.msdn.com/powershell/archive/2006/11/24/what-s-up-with-psbase-psextended-psadapted-and-psobject.aspx), chaque objet possède, renseignée ou non, les propriétés suivantes :

PSBASE	La vue brute de l'objet
PSADAPTED	La vue adaptée complète de l'objet
PSEXTENDED	Uniquement les membres étendus de l'objet
PSOBJECT	Une vue de l'adaptateur lui-même

On peut consulter à l'aide du logiciel Reflector le contenu des propriétés *AdaptedMemberSetName*, *BaseObjectMemberSetName* et *ExtendedMemberSetName* :



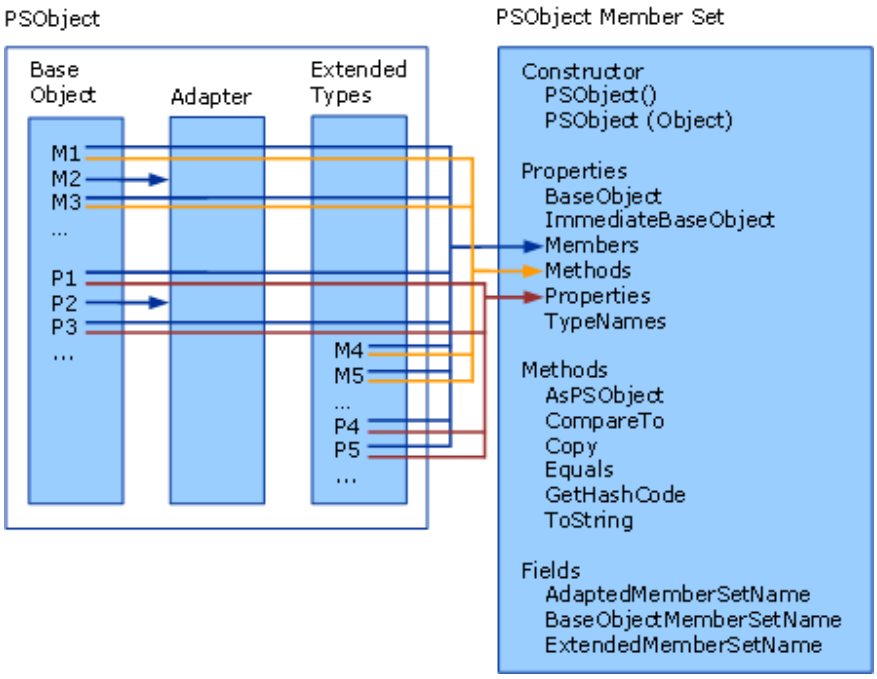
Détails de ces propriétés :

BaseObjectMemberSetName	Spécifie le nom de l'ensemble des membres de l'objet de base. Il s'agit des membres de l'objet de base encapsulé. Contient "***PSBase***".
AdaptedMemberSetName	Spécifie le nom de l'ensemble des membres adaptés. Contient "***PSAdapted***".
ExtendedMemberSetName	Spécifie le nom de l'ensemble des membres étendus. Ce sont les membres supplémentaires ajoutés à l'objet PSObject. Contient "***PSExtended***".

Ces trois propriétés sont présentes sur chaque objet que vous utilisez dans PowerShell. Elles sont invisibles et ne sont pas listées par **Get-member**.

Il existe également une propriété invisible nommée *PSTypeName*, elle présente le membre ***PSObject.TypeNames*** sous forme de collection, voir aussi :
http://keithhill.spaces.live.com/blog/cns!5A8D2641E0963A97!6034.entry

Sur le graphique suivant, issue du SDK, on peut voir que l'adaptateur masque les membres **P2** et **M2** de l'objet de base encapsulé dans une instance de **PSObject**.



Bien évidement les membres **P2** et **M2** restent accessibles mais en adressant l'objet de base

```
$x.psbase
```

Vous remarquerez, comme indiqué précédemment, que des membres peuvent être ajoutés à l'instance de **PSObject**. Un objet PowerShell peut donc être adapté et étendu mais dans tous les cas l'objet de base n'est pas modifié.

```
$p = get-wmiobject -class win32_computersystem
 #Affiche le no met le contenu des membres par défaut de
 #l'objet PowerShell
$p

 #Affiche tous les membres de l'objet PowerShell
$p|gm

 #Affiche tous les membres de l'objet de base (WMI)
$p.psbase|gm

 #Affiche tous les membres adaptés de l'objet PowerShell,
 #l'objet WMI est «transformé» en un objet .NET
$p.psadapted|gm
```

```
 #Affiche les membres étendu de l'objet PowerShell,
 #c'est-à-dire ajoutés à l'objet WMI
$p.psextended|gm
```

Note :

Si vous ajoutez un membre portant un nom identique à un membre de l'objet de base, le membre étendu aura la priorité sur le membre de l'objet de base.

2.2 PowerShell V2 : Evolutions concernant ETS

« *Improvements in Windows PowerShell V2 …*

Updated Type Adapters*: Member lookup algorithm for Type Adapters has been updated to include Base members. Members on the Base Object for Adapted types are now directly accessible on the object. This removes the need to use PSBASE property to access base members.*

Note :

L'expression « lookup algorithm » référence le fait que PS v1.0 recherche les membres en premier dans la liste des membres étendus d'une instance puis dans la liste des membres étendus du type (voir les fichiers .ps1xml) et enfin dans la liste des membres adaptés.

On peut supposer que cette évolution simplifie l'accès aux membres tout en l'accélérant.

Improvements to Get-Member*: Get-Member cmdlet now supports a –View and –Force parameter. Valid values for –View parameter are "Extended, Adapted, Base, All". The default value is "Extended,Adapted". Getter and Setter Methods are not shown by the default get-member output. Since adapted Types now expose Base members, use –view All to list Base members in the get-member output.*

Improved ADSI support*: ADSI adapter now allows access to Base Methods and Properties. [ADSISearcher] Type accelerator has been added for System.DirectoryServices.DirectorySearcher Class. Two CodeMethods, ConvertDNBinaryToString and ConvertLargeIntegerToInt64 have been added to DirectoryEntry type to simplify marshalling data between ADSI and PowerShell.*

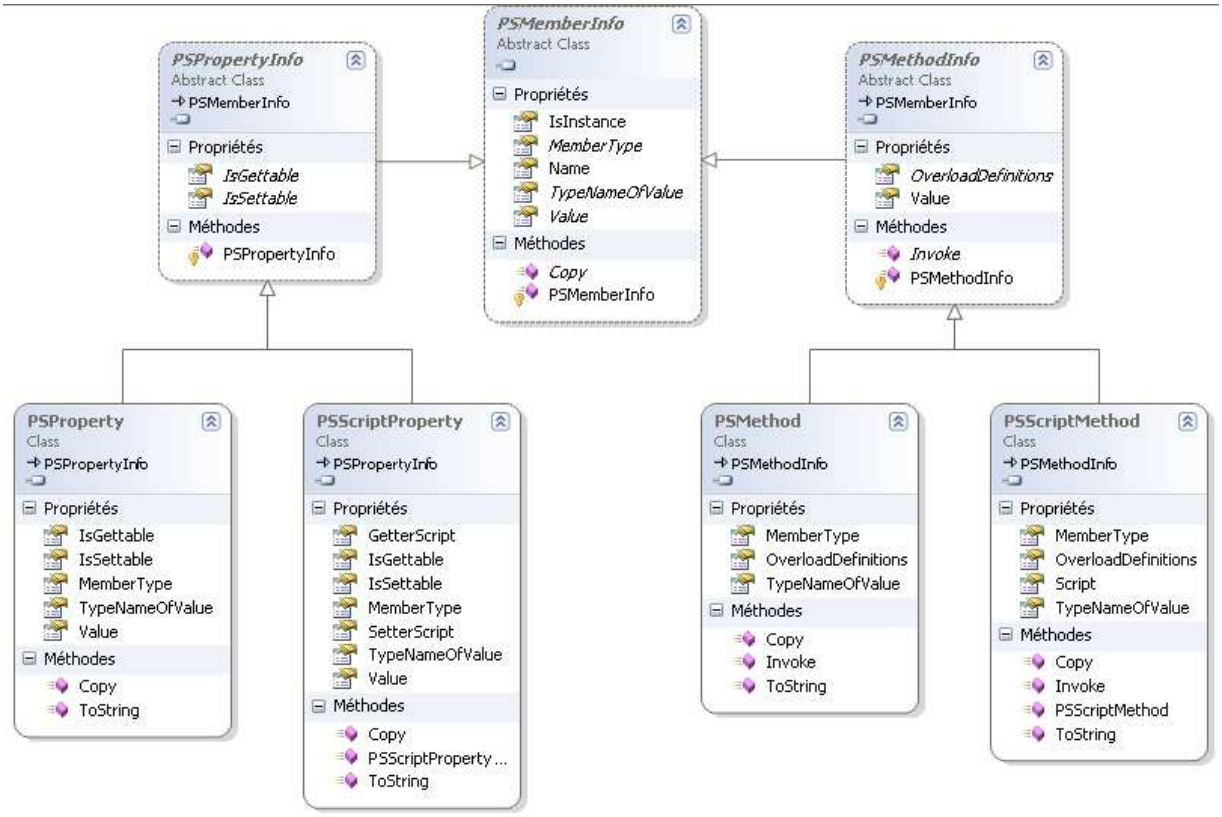
… »

Get-member propose dans la préversion 2 de PowerShell un paramètre supplémentaire nommé ***view*** :

```
$x=get-date
 #ps v1
$x.psbase | get-member
 #ps v2
$x | get-member –VIEW base
```

3 Les différents types de membres synthétiques

Tous les membres synthétiques héritent de la classe **PSMemberInfo** :



Le cmdlet **Add-Member** permet l'ajout de membres synthétiques, vous trouverez le détail de l'aide de ce cmdlet ici : http://msdn.microsoft.com/en-us/library/bb978596.aspx?s=6

Ces ajouts de membres ne portent que sur l'instance en cours et pas sur la classe de l'objet manipulé, en d'autres termes si vous souhaitez enrichir n objets vous devez effectuer le même traitement sur ces n objets.

Note :

Le cmdlet **Add-Member** ne fonctionne qu'avec des instances de type **PSOjbect :**

```
$i=10
$i|add-member NoteProperty Test 10
$i|gm
 #$i ne contient pas la nouvelle propriété
```

On doit caster l'objet pour effectuer l'ajout de membre:

```
$i=10 -as [PSObject]
$i|add-member NoteProperty Test 10
$i|gm
 #$i contient bien la nouvelle propriété
```

L'approche suivante est à privilégier :

```
$i=10
$i=$i|add-member NoteProperty Test 10 –passthru
$i|gm
```

Pour détailler les opérations de conversion vous pouvez utiliser le cmdlet **Trace-Command** :

```
Trace-Command TypeConversion {[psobject]$i=10} -psh
```

La documentation est donc erronée bien que sur MS-Connect un bug soit dans l'état *open* depuis 2006.

Voyons maintenant les différents types de membres à notre disposition.

3.1 AliasProperty

Définit un nouveau nom pour une propriété existante. L'ajout de ce membre permet de manipuler une information identique entre 2 classes dont les membres natifs, portant l'information, possèdent un nom différent. Ce type de propriété est donc en lecture/écriture.

Par exemple la classe **Array** propose la propriété *Length*, l'ajout d'un alias permet de disposer également d'une propriété fictive nommé *Count :*

```
$a=@(1,2)
$a.psextended|gm
   TypeName: System.Management.Automation.PSMemberSet
Name  MemberType    Definition
----  ----------    ----------
Count AliasProperty Count = Length

$a.psbase
Length         : 2

 #--------------
$o1=1|Select Nom
$o1.Nom="Premier"
$o2=1|Select Name
$o2.Name="Second"
$o1|Add-Member AliasProperty Name Nom
$o1,$o2|Foreach {$_.Name}
```

3.2 NoteProperty

Définit une propriété hébergeant une valeur ou un objet. La documentation parle de *propriété avec une valeur statique*. **Attention** l'objet hébergé peut être modifié, car ici il s'agit bien d'une instance d'une classe quelconque. Ce type de propriété est donc en lecture/écriture :

```
$Dt=Get-Date
   # Crée un objet de base
```

```
$o = new-object System.Management.Automation.PsObject
 #NoteProperty : Ajoute une note, c'est une propriété de type objet
$o|add-member NoteProperty DateCreation $Dt

$o
DateCreation
------------
31/08/2008 16:39:44

 #Modification du contenu par un objet de type Array
$o.DateCreation=dir
```

3.3 CodeMethod

Définit une méthode référençant une méthode statique d'une classe .NET compilée. Cette classe doit au préalable être chargée. Ce membre permet donc d'ajouter du code compilé à une instance d'objet PowerShell.

Ajoutons à une instance d'entier la fonction d'élévation à puissance donnée, [Math]::Pow :

```
$i=10
$M=[System.Math].GetMethod("Pow")
$i|add-member CodeMethod Puissance $M
```

```
Add-Member : Cannot create a code method because of the method format. The method should be public, static, and have one parameter of type PSObject.
```

Dans ce cas cette opération n'est pas possible car la méthode référencée par **Add-Member** doit être publique, statique et posséder au moins un paramètre de type PSObject. La signature de la méthode ***[System.Math].Pow*** est bien publique et statique mais le premier paramètre d'un membre de type *CodeMethod* doit être du type **PSObject**. Notez aussi que le code de la méthode référencée doit être thread-safe.

Un exemple de code C# :

```
namespace Synthetique
{
    public class MembresEtendus
    {
         //méthode privée
        static double ConvertExposant(PSObject ObjetPS)
        {
            if (ObjetPS.Properties["Exposant"].Value is PSObject)
             { return Convert.ToDouble((ObjetPS.Properties["Exposant"].Value
as PSObject).BaseObject); }

            return Convert.ToDouble(ObjetPS.Properties["Exposant"].Value);
        }
```

```
        public static double Puissance (PSObject ObjetPS)
        {
            Double x = Convert.ToDouble(ObjetPS.BaseObject);
            return Math.Pow(x, MembresEtendus.ConvertExposant(ObjetPS));
        }

        public static double ExposantGet (PSObject ObjetPS)
        {
            return MembresEtendus.ConvertExposant(ObjetPS);
        }

        public static void ExposantSet (PSObject ObjetPS,  double valeur)
        {
            ObjetPS.Properties["Exposant"].Value= valeur;
        }
    }
}
```

Le code PowerShell associé :

```
 #chargement de l'assembly contenant le code référencé
$DLLpath ="VotreRépertoire"
$Fullpath ="$DLLpath\Synthetique.dll"
[void][Reflection.Assembly]::LoadFile($Fullpath)

 # déclaration du membre étendu de type CodeMethod
$i=5
$i=$i|add-member NoteProperty Exposant ([int]3) -passthru;$i|gm
 #Obtient une référence sur la méthode nommée "Puissance"
$M=[Synthetique.MembresEtendus].GetMethod("Puissance")
$i=$i|add-member CodeMethod Puissance $M -passthru;$i|gm
$i.Puissance()
```

Vous trouverez dans le chapitre **Liens** une méthode pour débugger ce type de code sous PowerShell et VisualStudio 2005.

3.4 CodeProperty

Définit une une propriété référençant référençant une propriété statique d'une classe .NET compilée. Cette classe doit au préalable être chargée. Ce membre permet donc d'ajouter du code compilé à une instance d'objet PowerShell. Il permet de déclarer des accesseurs (getter et setter) déclarés dans un langage compilé.

Son usage le plus courant semble être pour une propriété calculée.

Nous utiliserons le même code que pour le membre de type **CodeMethod** :

```
namespace Synthetique
{
```

```
    public class MembresEtendus
    {
        …
        public static double ExposantGet (PSObject ObjetPS)
        {
            return MembresEtendus.ConvertExposant(ObjetPS);
        }

        public static void ExposantSet (PSObject ObjetPS,  double valeur)
        {
            ObjetPS.Properties["Exposant"].Value= valeur;
        }
    }
}
```

Le code PowerShell associé :

```
$o=1|select Valeur
$o.Valeur=10
$o|add-member NoteProperty Exposant 3 –passthru
$getter=[Synthetique.MembresEtendus].GetMethod("ExposantGet")
$setter=[Synthetique.MembresEtendus].GetMethod("ExposantSet")
$o| add-member CodeProperty MaPropriete $getter $setter;$o|gm
$o.MaPropriete=5
$o
```

Il est possible de déclarer une telle propriété en lecteur seule, le setter étant $null ou n'étant pas renseigné :

```
$o| add-member CodeProperty MaPropriete $getter
```

Dans ce cas l'accès en écriture lève une exception :

```
$o.MaPropriete=10
```

```
Set accessor for property "MaPropriete" is unavailable.
```

Cet exemple de code C# n'est pas d'une grande utilité et est réduit à sa plus simple expression, notez toutefois les possibles conversions et contrôles à effectuer.

3.5 PropertySet

Définit une collection de propriétés appartenant à un objet. Ce n'est pas une énumération mais une liste de propriétés pouvant être utilisé avec le cmdlet **Select-Object** ou tout autre cmdlet utilisant une liste de propriétés en tant que paramètre. Ce membre permet de filtrer certaines propriétés lors de traitements spécifiques :

```
 #On récupère le contenu du répertoire courant dans un tableau de fichiers
$F=Get-item *.*
  #On crée un tableau de nom de propriétés
$F|add-member PropertySet "PSProperties"
([string[]]("PSChildName","PSDrive","PSIsContainer","PSParentPath","PSPath
","PSProvider"))
  #On affiche une vue personnelle de l'objet
$F[2]| select PSProperties
```

Un exemple complet d'utilisation de ce membre :

http://mow001.blogspot.com/2006/01/some-fun-with-monads-add-member-mp3.html

Certains objets adaptés peuvent posséder de telles propriétés :

```
$p = get-wmiobject -class win32_computersystem
$p.Power
$p.Power|gm
$p|Select Power
```

3.6 MemberSet

Définit une collection de membres (propriétés et méthodes) appartenant à un objet. Cela permet de déclarer un sous-ensemble de membres utilisé par des traitements particuliers.

Par exemple le memberset *PSStandardMembers* permet de spécifier, au travers de l'élément *DefaultPropertySet*, l'ensemble des méthodes et propriétés affichées en standard :

```
$o=1|Select Nom, DureeDeVie, Numero
$DefaultProperties =@('Nom','DureeDeVie')
$DefaultPropertySet=New
System.Management.Automation.PSPropertySet('DefaultDisplayPropertySet',[st
ring[]]$DefaultProperties)
$PSStandardMembers=[System.Management.Automation.PSMemberInfo[]]@($Default
PropertySet)
$o|Add-Member MemberSet PSStandardMembers $PSStandardMembers
```

Exemple issu du site :

http://poshoholic.com/2008/07/05/essential-powershell-define-default-properties-for-custom-objects/

On peut donc déclarer sur tout objet ce type de collection pour faciliter certains traitements. En revanche ces *memberset* peuvent ne pas tous être modifiable :

```
$o.PSStandardMembers.DefaultDisplayPropertySet.IsInstance
```

```
True
 #C'est à dire que ce membre ne provient pas d'un fichier de configuration .ps1xml
```

```
$f=dir|select -last 1
$f.PSStandardMembers.DefaultDisplayPropertySet.IsInstance
```

```
False
#C'est à dire que ce membre provient d'un fichier de configuration .ps1xml
```

```
$o.PSStandardMembers.DefaultDisplayPropertySet.ReferencedPropertyNames.A
dd('Numero')
$o
```

```
Nom      DureeDeVie                Numero
---      ----------                ------
```

```
[void]$o.PSStandardMembers.DefaultDisplayPropertySet.ReferencedPropertyNam
es.remove('Nom')
$o
```

```
DureeDeVie                 Numero
```

```
----------                    ------
```

Pour la variable **$f** cela ne fonctionnera pas car son membre *DefaultDisplayPropertySet* n'est pas une instance mais est ajouté via le fichier **types.ps1xml** :

```
$f.PSStandardMembers.DefaultDisplayPropertySet.ReferencedPropertyNames.A
dd('Attributes')
$f
```

```
   Directory: Microsoft.PowerShell.Core\FileSystem::G:\test
Mode         LastWriteTime      Length      Name
…
```

Ceci est confirmé pas la tentative de remplacement de ce *memberset* :

```
$f|Add-Member MemberSet PSStandardMembers $Nouveau -force
```

```
Add-Member : Cannot force the member with name "PSStandardMembers" and type "MemberSet" to be added
because a member with that name and type already exists and the existing member is not an instance extension.
```

3.7 ParameterizedProperty

L'ajout d'un membre de ce type provoquera l'erreur suivante :

```
$o | Add-Member ParameterizedProperty PP {[DateTime]::Now-
$this.DateCreation}
```

```
Impossible d'ajouter un membre du type « ParameterizedProperty ». Spécifiez un autre type pour le paramètre
MemberTypes.
```

Ce type de membre ne peut être ajouté via **Add-Member** car les propriétés paramétrées sont des propriétés paramétrées par COM et sont donc créées seulement par des adaptateurs. Ce type de propriété n'est pas directement utilisable par le langage de script PowerShell.

Voir : http://msdn2.microsoft.com/en-us/library/cc136162(VS.85).aspx

3.8 ScriptMethod

Définit une méthode, son contenu est un script PowerShell. Il est possible d'utiliser des paramètres :

```
$o|add-member -force -membertype scriptmethod MaMethodePS_V1{
    Write-host $args[0]
    Write-host $args[1]
   "Arguments =$args"
   }
```

L'usage de la clause **Param** ne fonctionne pas sous PowerShell v1 :

```
$o|add-member -force -membertype scriptmethod MaMethodePS_V2{
   param([int] $Valeur, [String] $Nom)
    Write-host $Valeur
    Write-host $Nom
    "Arguments =$args"
   }

$o.MaMethodePS_V1(10,"Test méthode")
```

```
$o.MaMethodePS_V2(10,"Test méthode")
```

PowerShell ne propose pas de surcharge de méthode il reste possible de la simuler à l'aide d'un membre de ce type, voir le post suivant :
http://www.wiredbox.net/Forum2/Thread11798_IDEA_scriptmethod_method_overload_(and_functions_too).aspx

3.9 ScriptProperty

Définit une propriété dont le contenu est une valeur ou le résultat d'un bloc de script. Ce membre peut être utilisé pour des propriétés calculées :

```
$Dt=Get-Date
$o = new-object System.Management.Automation.PsObject
$o| add-member NoteProperty DateCreation $Dt
$o | add-member -memberType Scriptproperty -Name DureeDeVie -value
{[DateTime]::Now-$this.DateCreation} -SecondValue {write-host "Erreur:
Cette propriété est en lecture seule."}
  #Le type de donnée est un interval de temps, TimeSpan
$o.DureeDeVie
$o.DureeDeVie.ToString()
```

Ce type de propriété posséde des accesseurs, les paramètres *–Value* et *–SecondValue* étant respectivement le *getter* et le *setter*. Dans cet exemple on construit une propriété en lecture seule. L'affection d'une valeur généra une exception.

```
$o.DureeDeVie=10
```

```
Erreur: Cette propriété est en lecture seule.
```

La variable locale **$this** référence l'objet courant c'est-à-dire **$O**.

3.10 Ajout de membre via la liste des membres

Il est tout à fait possible d'ajouter des membres sans utiliser le cmdlet **Add-Member** :

```
$objet=1|Select Nom

$Propriété = New-Object System.Management.Automation.PSNoteProperty
'Date', (Get-Date)
$objet.PSObject.Properties.Add($Propriété)

$Setter={$this.Date.DayOfweek}
 #Propriété en lecture seule
$Membre=new-object management.automation.PSScriptProperty "Jour", $Setter
$objet.PSObject.Members.Add($Membre)
$Objet
```

3.11 Ajout de membre via un fichier xml

ETS permet, à la différence du cmdlet **Add-Member**, d'ajouter des membres synthétiques pour chaque instance d'une classe donnée. Pour plus d'informations consultez les liens suivants :

- Add Custom Methods and Properties to Types in PowerShell :

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

- Extending Object Types and Formatting

http://msdn.microsoft.com/en-us/library/cc136149(VS.85).aspx

- Format XML Schema

http://msdn.microsoft.com/en-us/library/cc136082(VS.85).aspx

Voir aussi le fichier **about_types.ps1xml.help.txt** présent dans le répertoire de PowerShell.

4 Exemple de création d'un objet synthétique

Je vous propose comme objectif de créer un objet séquence similaire à un objet séquence d'un SGBDR, plus particulièrement celui d'Oracle.

Le rôle d'une séquence est de fournir une suite de nombres entiers, suite déterminée par la valeur de certains paramètres.

Pour construire un tel objet nous créerons une méthode et des propriétés scriptées en lecture seule afin d'interdire la modification des données utilisées en interne par la code de notre méthode.

4.1 Remarques et limites des object synthétiques

Etant donné que PowerShell n'est pas orienté objet, c'est-à-dire qu'il ne propose pas d'instruction de création de classe, la création d'objet personnalisé comporte quelques limites :

-définition de classe impossible, tous les objets personnalisés sont de la même classe (PSCustomObject), sauf bien évidemment les objets .NET tel que FileInfo, Integer…,

-déclaration de champs privés inexistante, toutes les déclarations de membres synthétiques sont accessibles en lecture et en écriture par l'utilisateur de votre objet,

-définition d'interface impossible,

- la surcharge de méthode n'existe pas mais peut être simulée pour ne méthode,

- la création de générique reste difficile, comme d'autres opérations .NET.

4.2 Définition de type d'un objet personnalisé

Ne disposant pas de mécanisme de création de classe il reste un problème à régler. Puisque tous les objets personnalisés sont toujours de la même classe (PSCustomObject), le type de la classe étant immuable, comment déterminer si un ou n objets sont d'un type particulier ?

Afin de tester si l'objet personnalisé est d'un type particulier on peut ajouter une propriété contenant un GUID, un identifiant unique, on simulera ainsi un nom de type unique pour le ou les objets personnalisés :

```
add-member ScriptProperty ClassGuid -value {"18e85e6f-0982-4b8eb74f-
0a096fc44436"}
```

Pour s'assurer que l'objet manipulé est bien un objet personnalisé on testera la propriété GUID :

```
if ( ($Item -is [System.Management.Automation.PSObject] )
-and
($Item.ClassGuid -eq "18e85e6f-0982-4b8e-b74f-0a096fc44436")
)
```

L'inconvénient étant de mémoriser la correspondance entre le GUID et l'objet personnalisé.

Une autre approche consiste à ajouter dynamiquement un nom de type à la collection **TypeNames** :

```
$O=1|select nom
 #Ajoute un nom de type
$O.PSObject.TypeNames.Insert(0,'Sequence')
 #Le type réel de l'objet n'est pas modifié
$O.GetType()
```

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

```
$O.PSTypeNames
```

```
Sequence
Selected.System.Int32
System.Management.Automation.PSCustomObject
System.Object
```

```
$o.PSObject.TypeNames -contains "Sequence"
```

```
True
```

Dans ce cas il faut mémoriser la liste des noms de type que vous utilisez afin d'éviter de réattribuer ces noms à d'autre type d'objet personnalisé.

(Origine :

http://poshoholic.com/2008/07/03/essential-powershell-name-your-custom-object-types/)

4.3 Déclaration des données de l'objet personnalisé

Lors de la déclaration d'un objet personnalisé au sein d'un script on peut rencontrer un problème, la portée des variables :

```
Function Test
{ param([String] $Comment)

 $Objet= new-object System.Management.Automation.PsObject
 $Objet | add-member ScriptProperty Commentaire  -value {$Comment}
```

```
 Write-host ("*** Debug interne : {0}" -F $Objet.Comment)
 $Objet
}
$Comment=$null
 #Possible effet de bord
#$Comment="Variable globale"

$O=Test "Paramètre"
$O
```

Ici le membre *Commentaire* reste vide car la variable *$Comment*, c'est à dire le paramètre, n'est plus accessible.

Si on supprime le commentaire de la seconde déclaration on a droit à un petit effet de bord car le nom de la variable référencée, *$Comment*, existe dans la portée parente.

Pour éviter ceci on peut utiliser un membre de type NoteProperty :

```
Function Test
{param([String] $Comment)

 $Objet= new-object System.Management.Automation.PsObject
 $Objet | add-member NoteProperty Commentaire  $Comment
 Write-host ("*** Debug interne : {0}" -F $Objet.Comment)
 $Objet
}
$Comment="Variable globale"

$O=Test "Paramètre"
$O
$Comment
```

Dans ce cas le problème de portée n'existe plus mais l'accès en écriture sur le membre *Commentaire* est désormais possible :

```
$O.Commentaire=10
$O
```

Selon les cas on peut se contenter de cette situation.

Essayons une autre approche :

```
Function Test
{ param([String] $Comment)

 $Objet= new-object System.Management.Automation.PsObject
 $Objet | add-member NoteProperty Comment  $Comment
```

```
 $Objet | add-member ScriptProperty Commentaire  {$this.Comment}
 $Objet
}
```

Cette construction pourrait régler le problème mais malheureusement PowerShell n'offre pas de possibilité de masquer le membre *Comment.* Ou tout du moins de restreindre son accès en lecture seule. Nous verrons plus loin comment créer des membres en lecture pour l'utilisateur de l'objet et en lecture/écriture au sein d'une méthode d'un objet personnalisé.

Il est possible retarder l'accès au membre *Comment*, comme ceci :

```
Function Test
{ param([String] $Comment)

  #Tableau des variables utilisées par l'objet
 $Data=@{Comment=$Comment};
 $Objet= new-object System.Management.Automation.PsObject
 $Objet | add-member NoteProperty ("C "+[char]8+"omment") -value $Data
 $Objet | add-member ScriptProperty Commentaire  {$this.("c "+[char]8+"omment").Comment}
 $Objet
}
$O=Test "Accès retardé"
$O
```

```
Data                Commentaire
------              -----------
{Comment}           Accés retardé
```

```
$O.Comment
$O.Comment|gm
```

```
Get-Member : No object has been specified to get-member.
```

J'ai dit qu'il était possible de retarder l'accès au membre car l'affichage par **Get-Member** révélera l'astuce…

4.4 Vérification des paramètres d'un script ou d'une fonction

Si on souhaite coder une analyse rigoureuse des paramètres de la ligne de commande, notamment savoir si un paramètre est mal orthographié ou inconnu, l'objet contextuel *$MyInvocation* peut nous être utile :

Fonction sans contrôle particulier :

```
Function Test-UnBoundArguments
{ param([String] $Name,[switch] $Stop)
 #Suite du traitement...
}
```

```
Test-UnBoundArguments
Test-UnBoundArguments "Test"
Test-UnBoundArguments -name "Test"
Test-UnBoundArguments -ParamInconnu 10
Test-UnBoundArguments -nime "Test"
Test-UnBoundArguments -Stop
Test-UnBoundArguments –SwitchInconnu
```

```
# Pas d'erreur
```

Fonction contrôlant les paramètres inconnus, i.e. "non lié" à l'aide la collection *$MyInvocation.UnBoundArguments* :

```
Function Test-UnBoundArguments
{ param([String] $Name,[switch] $Stop)


 if (($MyInvocation.UnBoundArguments).count -ne 0)
   {Throw "Le ou les paramètres suivants sont inconnus :
$($MyInvocation.UnBoundArguments)."}
 #Suite du traitement ...
}


Test-UnBoundArguments
Test-UnBoundArguments "Test"
Test-UnBoundArguments -name "Test"
Test-UnBoundArguments -ParamInconnu 10
Test-UnBoundArguments -nime "Test"
Test-UnBoundArguments -Stop
Test-UnBoundArguments -SwitchInconnu
```

```
Le ou les paramètres suivants sont inconnus : -SwitchInconnu.
```

4.5 Vérification de la présence des paramètres contraints

Un autre problème peut se poser avec les paramètres contraints ; Comme ils sont initialisés à une valeur par défaut, par exemple pour les entiers à zéro, on ne peut donc pas les tester sur la valeur *$null* afin de déterminer si un paramètre est présent ou non sur la ligne de commande.

Parfois la valeur attribuée par défaut peut ne pas être souhaitable, dans ce cas on testera la collection *$MyInvocation.CommandLineParameters* qui contient la liste des paramètres déclarés par dans la clause ***Param*** du script en cours d'exécution :

```
Function Test-UnBoundArguments
{ param([String] $Name,[int]$Nombre,[switch] $Stop)


 if (($MyInvocation.CommandLineParameters.Nombre -eq $null))
  {Write-Warning "Le paramètre Nombre n'est pas précisé."}
"Nombre =$Nombre"
```

```
}

Test-UnBoundArguments -nombre 5
Test-UnBoundArguments -nombre
Test-UnBoundArguments
```
WARNING: Le paramètre Nombre n'est pas précisé.

Si on déclare une valeur par défaut, pour le paramètre *$Nombre*, le test renverra *$true* mais la affectée sera bien celle précisée et non pas zéro.

4.6 Déclaration d'un membre d'un objet personnalisé en accès privé

La syntaxe de création de membres synthétiques ne permet pas de créer des champs privés accessibles uniquement par le code de l'objet personnalisé.

Il y a bien le membre *ScriptProperty* qui permet de ne pas déclarer de setter (accesseur en écriture) mais dans ce cas on ne déclare rien d'autre qu'une constante :

```
$O= new-object System.Management.Automation.PsObject
$O|add-member ScriptProperty Comment -value {"Commentaire en readOnly"}
$O.Comment="Nouveau contenu"
```

Il reste possible de redéclarer l'objet avec un nouveau contenu mais dans ce cas on perd toutes les valeurs des possibles autres membres à moins de recréer chaque membre à l'identique.

De plus comme nous l'avons vue si on utilise un membre de type *NoteProperty* pour porter notre donnée ce membre reste accessible en lecture/écriture.

Pour un membre de type *ScriptProperty* ou *ScriptMethod*, le paramètre *value* de Add-member est un scriptbloc, on peut donc y utiliser un type valeur et s'appuyer sur le mécanisme du pipeline :

```
$O| add-member ScriptProperty Min -value {10}
 # Peut être :
 # $O| add-member ScriptProperty FirstFile -value {Dir|Select -First 1}
$O.Min
$O.Min=11
```

Puisqu'on ne souhaite utiliser ni un membre de type *NoteProperty* ni de type *ScriptProperty* on peut obtenir le comportement attendue en construisant le code de la déclaration du membre synthétique. C'est-à-dire utiliser les possibilités du langage dynamique de PowerShell afin de définir ou redéfinir du code à la volée. Ce code généré n'existant pas dans le script d'origine, le seul code qui existe est un code de création de code…

Voyons comment cela fonctionne.

4.7 Un peu de dynamisme

On déclare une chaîne de caractères paramétrée, i.e. utilisée avec l'instruction de formatage **-F**, contenant la définition de notre membre synthétique :

```
$RazMember="`$O| add-member -Force ScriptProperty Min -value {0}"
```

La variable *$RazMember* référence l'objet concerné (on pourrait aussi le paramétrer) nommé $O.

Ensuite on récupère la chaîne formatée en lui fournissant la valeur aue l'on attribuera au paramètre –*value* du cmdlet **Add-Member** :

```
$MaValeur=257
$S=$RazMember -F "{[int]$MaValeur}"
```

Ce qui génère la chaîne suivante :

```
 $O| add-member -Force ScriptProperty Min -value {[int]257}
```

Notez qu'on précise le type de la valeur.

Enfin on exécute le code contenu dans notre chaîne de caractères :

```
Invoke-Expression $S
```

L'accès au membre ainsi créé renvoie bien la nouvelle valeur :

```
$O.Min
```

Le membre concerné a été modifié mais reste en lecture uniquement :

```
$O.Min=-1
```

```
Set accessor for property "Min" is unavailable.
```

Une fonction d'exemple :

```
Function Create-SyntheticMemberRO
{ param([String] $Comment,
       [int] $MaxValue,
       [switch] $Cycle)

#Création de l'objet
 $Objet= new-object System.Management.Automation.PsObject

 # Ajout des propriétés synthétiques en R/O, le code est créé
dynamiquement
$MakeReadOnlyMember=@"
`$Objet | add-member ScriptProperty Comment -value {"$Comment"} -Pass|`
    add-member ScriptProperty MaxValue -value {[int]$MaxValue}  -Pass|`
    add-member ScriptProperty Cycle -value {`$$Cycle}
"@
  Invoke-Expression $MakeReadOnlyMember

 $Objet| add-member ScriptMethod Inc{
  $NewValue=$this.MaxValue
  $NewValue++
  #Déclaration paramétrée pour la redéfinition du membre CurrVal
  #Le paramètre -Force annule et remplace la définition du membre spécifié
  $RazMember="`$this|add-member -Force ScriptProperty MaxValue -value {0}"
```

```
  #On construit (par formatage) la définition du membre CurrVal
  #puis on reconstruit le membre.
  Invoke-Expression ($RazMember -F "{[int]$NewValue}")
                        }
  #Spécifie l'affichage des propriétés par défaut.
  #On évite ainsi l'usage  d'un fichier de type .ps1xml
 $DefaultProperties =@(
  'MaxValue',
  'Cycle',
  'Comment'
  )
 $DefaultPropertySet=New
System.Management.Automation.PSPropertySet('DefaultDisplayPropertySet',`
[string[]]$DefaultProperties)
 $PSStandardMembers=[System.Management.Automation.PSMemberInfo[]]`
@($DefaultPropertySet)
 $Objet|Add-Member MemberSet PSStandardMembers $PSStandardMembers

 return $Objet
}

$Obj=Create-SyntheticMemberRO "Membre en lecture seule" 25 -Cycle
$obj
$obj.MaxValue
 #Modification impossible
$obj.MaxValue=10
$obj.Inc()
$obj
```

Ainsi on peut donc, peut être au détriment des performances, créer des membres synthétiques en lecture seule. Bien évidemment comme il reste possible de supprimer les membres d'un objet personnalisé cette solution ne répond pas entièrement à notre problème de départ.

Gardez à l'esprit que ce mécanisme n'est pas à utiliser à tout bout de champs sinon il risque fort de se transformer en une dynamite du langage ;-)

De plus cette approche peut ne pas s'appliquer dans tous les cas mais le C# ou tout autre langage .NET simplifiera et réglera le problème.

4.8 L'objet séquence

Le script *Create-Séquence.ps1* ne contient rien d'autres de particuliers si ce n'est les règles et les contrôles propre à la gestion d'une séquence qui sont suffisamment documentés.

Voici quelques exemples de séquences :

```
$N="SEQ_Test"
$C="Séquence de test"

Sq=Create-Sequence $N

 #suite ascendante débutant à 0 et finissant à 255
$Sq=Create-Sequence $N $C -min 0 -max 255

 #suite cyclique ascendante débutant à 5, ensuite chaque cycle
 #débutera à 0
$Sq=Create-Sequence $N $C -min 0 -max 5 -inc 1 -start 5 -cycle

 #Suite ascendante maximum, de [int32]::MinValue à [int32]::MaxValue
 #débutant à [int32]::MinValue
$Mini=([int32]::MinValue+1)
$Sq=Create-Sequence $N $C -min $Mini -start $Mini

 #Suite descendante maximum, de [int32]::MinValue à [int32]::MaxValue
 #débutant à [int32]::MaxValue
$Maxi=([int32]::MaxValue)
$Sq=Create-Sequence $N $C -min $Mini –max $Maxi -start $Maxi -inc -1
```

La visualisation de l'objet affichera ceci :

```
$Sq
```

```
Name          : SEQ_Test
CurrVal       : 2147483647
Increment_By  : -1
MaxValue      : 2147483647
MinValue      : -2147483647
Start_With    : 2147483647
Cycle         : False
Comment       : Séquence de test
```

La récupération de la valeur courante de la séquence se fait *via* la propriété *CurrVal* :

```
$Sq.Currval
```

```
CurrVal       : 2147483647
```

L'obtention de la prochaine valeur de la séquence se fait *via* la méthode *NextVal* :

```
[void]$Sq.NextVal()
$Sq.Currval
```

```
CurrVal       : 2147483646
```

Ou

```
$Sq.NextVal()
```

```
2147483645
```

```
$Valeur=Sq.NextVal()
$Valeur
```

```
2147483644
```

5 Liens

Création d'objet, performance entre Select-Object, Add-Member et le C#

http://karlprosser.com/coder/2008/06/12/generating-a-propertybag-aka-pscustomobject-in-c/

Building and Debugging Powershell cmdlets in the VS IDE

http://blogs.msdn.com/jmstall/archive/2007/03/04/debugging-cmdlets.aspx

Un exemple plus évolué autour de Add-Member : Get Extended Properties of a File

http://keithhill.spaces.live.com/blog/cns!5A8D2641E0963A97!186.entry

Autres approches

PowerShell permet la création de grammaire ou DSL (Domain-Specific Language), en utilisant le dynamisme et/ou les membres synthétiques.

Dans le livre « PowerShell in action » l'auteur aborde, dans le chapitre 8, rapidement ce sujet au travers du script Class.ps1 et Demo-Class.ps1

```
CustomClass point {
    note x 0
    note y 0
    method ToString {
        "($($this.x), $($this.y))"
    }
    method scale {
        $this.x *= $args[0]
        $this.y *= $args[0]
    }
}
```

Vous devrez peut être supprimé dans votre session PS l'alias ***new*** car ce code d'exemple déclare une fonction portant le même nom.

http://www.manning.com/payette/WPA_Example_Code.zip

Introduction aux DSL Tools sous VisualStudio 2005

http://www.dslfactory.org/blogs/alain/archive/2006/11/03/46.aspx

About Domain-Specific Languages

http://msdn.microsoft.com/en-us/library/bb126278.aspx

Using a DSL to generate XML in PowerShell

http://blogs.msdn.com/powershell/archive/2007/06/18/using-a-dsl-to-generate-xml-in-powershell.aspx

Extension Methods in Windows PowerShell (PS et les méthodes d'extensions du C# 3.5)

http://bartdesmet.net/blogs/bart/archive/2007/09/06/extension-methods-in-windows-powershell.aspx

Object Oriented Scripting in Powershell

http://cashfoley.com/2008/02/17/ObjectOrientedScriptingInPowershell.aspx