Usage des classes génériques avec Powershell

Par Laurent Dardenne, le 20/12/2021.

Conçu avec Powershell v5.1 Windows 10 64 bits.

Site de l'auteur : http://laurent-dardenne.developpez.com/
Sources des démos :
https://github.com/LaurentDardenne/Tutorial/tree/master/UsageDesClassesGénériquesAvecPowershell/Sources

Chapitres

1 Un type à part

On utilise souvent la classe [System.Array] afin de créer des tableaux :

```
$T=@(1,'un',10.5)
```

Dans cet exemple il contient plusieurs objets de type différent, dans ce cas le type de ce tableau est du type *System.Object[]* :

```
$T.GetType().Fullname
#System.Object[]
```

On peut vouloir préciser un type lors de la création d'un tableau , par exemple un tableau d'entier :

```
[Int[]] $T=1..5
$T.GetType().Fullname
#System.Int32[]
```

Ou encore un tableau de chaîne de caractères :

```
[String[]] $T=@('Un','Deux')
#System.String[]
```

Ceci pour nous contraindre à respecter le type des objets que l'on peut insérer, cela évite d'ajouter un objet d'un autre type que celui demandé, pour le langage C# ce contrôle se fait lors de la compilation et lors de l'exécution.

Pour PowerShell ce contrôle se fait lors de l'exécution :

```
[Int[]] $T +='Texte'
Impossible de convertir la valeur «Texte» en type «System.Int32».
Erreur: «Le format de la chaîne d'entrée est incorrect.»
```

Notez que l'affectation suivante fonctionne :

```
[Int[]] $T +='6'
```

PowerShell converti implicitement la chaîne en un entier. Dans l'exemple suivant on autorise uniquement des fichiers :

```
[System.IO.FileInfo[]] $Fichiers=Dir -File
$Fichiers += Get-Item $Env:Windir
Impossible de convertir la valeur « C:\WINDOWS » du type
« System.IO.DirectoryInfo » en type « System.IO.FileInfo ».
```

L'affectation qui suit réussie du fait d'une conversion implicite de PowerShell :

```
$Fichiers += 'Test'
[System.IO.FileInfo]::New('Test')
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
darhsl       01/01/1601     01:00              0 test
```

A noter qu'il est possible de typer un tableau avec nos propres classes.

Il faut savoir qu'en interne DotNet génère, avec le type qu'on lui précise, une classe spécifique dérivée de *System.Array* :

```
$T.PSTypeNames
#System.String[]
#System.Array
#System.Object
```

A aucun moment nous n'avons déclaré ici de classe en utilisant le langage C#.

La documentation de la classe Array du Framework 2.0 précise :

> *La classe Array est la classe de base pour les implémentations de langage qui prennent en charge des tableaux.* ***Toutefois, seuls le système et les compilateurs peuvent dériver explicitement de la classe Array.*** *Les utilisateurs doivent employer les constructions de tableau fournies par le langage.*

Si on souhaite créer une classe proposant le comportement d'une pile (*Stack*) pour tous les types, qui serait similaire à ce que permet les déclarations de tableau, il va nous falloir un mécanisme supplémentaire.

Il faut garder à l'esprit que le code des traitements d'une telle classe doit s'adapter à chaque type que l'on précisera. Ceci est possible grâce aux classes génériques.

2 Classe générique

La déclaration d'une classe générique en C# se base sur une syntaxe particulière, à savoir un nom de classe, suivi du caractère '<', puis d'un ou plusieurs noms de paramètre séparés par une virgule et enfin terminé par le caractère '>'.

Un exemple de déclaration d'un nom de classe générique déclarant un paramètre :
*MaClasseGenerique<**UnType***>

Plus précisément :
« Pour une classe générique Node<T>, le code client peut référencer la classe soit en spécifiant un argument de type, pour créer un type construit fermé (Node<int>), soit en laissant le paramètre de type non spécifié, par exemple quand vous spécifiez une classe de base générique, pour créer un type construit ouvert (Node<T>). »

Note : le vocabulaire est ardu car les spécifications d'un langage doivent être précises.

Une classe générique doit être compilée avant de pouvoir créer une classe spécifique :

```
$code=@'
using System;
public class ClasseGenerique<UnType>
{
    UnType Data;

    public ClasseGenerique(UnType Parametre)
    { this.Data = Parametre; }

    public void GetTypeInformation()
    {
       string S=typeof(UnType).FullName;
       Console.WriteLine(String.Format("Type = {0}",S));
    }
}
'@
Add-Type $code

$InfoInt = [ClasseGenerique[String]]::new('Test')
```

Une classe générique permet de paramétrer les types sur lesquels elle opère, ici le type ***String***.
Dans la déclaration de cette classe générique, ***UnType*** est un paramètre précisant un nom de type :

```
class ClasseGenerique<UnType>
```

Dans l'appel du constructeur, ***String*** est la valeur de ce paramètre :

```
[ClasseGenerique[String]]::New
```

Lors de la compilation les occurrences de ***UnType*** sont remplacées par le type **[String]**.

Le plus souvent dans une déclaration de classe générique les concepteurs utilisent des lettres pour spécifier des arguments de type, ici **T** :

```
$code=@'
using System;
public class MaClasseGenerique<T>
{
    T Data;
    public MaClasseGenerique(T Parametre)
     { this.Data = Parametre;  }

    public void GetTypeInformation()
     {
       string S=typeof(T).FullName;
       Console.WriteLine(String.Format("type = {0}",S));
     }
}
'@
Add-Type $code
```

C'est l'utilisateur du code qui paramètre la classe générique, et une fois celle-ci paramétrée c'est le compilateur qui se chargera d'abord de créer la classe générique (*type construit ouvert)*, ensuite il sera possible de créer des instances de cette classe générique (*type construit fermé)* :

```
$InfoInt = [ClasseGenerique[String]]::new('Test')
$InfoInt.GetTypeInformation()
type = System.String

$InfoInt=[MaClasseGenerique[Int]]::new(10)
$InfoInt.GetTypeInformation()
type = System.Int32
```

On ne peut pas insérer de chaîne de caractères dans une collection générique d'entiers :

```
$InfoInt=[MaClasseGenerique[Int]]::new('Texte')
Impossible de convertir l'argument «Parametre» (valeur «Texte») de
«.ctor» en type «System.Int32»:
```

On peut également visualiser le paramétrage de la classe générique à l'aide de la définition du constructeur :

```
[MaClasseGenerique[Int]]::new
OverloadDefinitions
-------------------
MaClasseGenerique[int] new(int Parametre)
[MaClasseGenerique[String]]::new
MaClasseGenerique[string] new(string Parametre)
```

Voir aussi la terminologie relative aux génériques. Nous aborderons par la suite quelques-uns de ces termes. Quant à notre souhait de créer une classe proposant le comportement d'une pile (*Stack*) pour tous les types, le Framework propose cette classe générique :

```
$Stack=[System.Collections.Generic.Stack[Int]]::new([int[]]@(1,3,5))
$Stack.Pop()
```

2.1 Retrouver les classes génériques disponibles

Pour retrouver les classes génériques d'un assembly, il nous faut soit interroger un objet de type *RuntimeAssembly :*

```
[AppDomain]::CurrentDomain.GetAssemblies().ExportedTypes
```

soit interroger n'importe quel nom de type, générique ou pas, ici le type générique ***List*** :

```
[System.Collections.Generic.List[int]].Assembly.ExportedTypes|
Where-Object {
   $_.IsPublic -and $_.IsGenericType -and $_.IsInterface -eq $false -
and $_.IsNested -eq $false -and
$_.IsSubclassOf([System.MulticastDelegate]) -eq $false
 }|
 Foreach-Object {"$_"}
System.ArraySegment`1[T]
System.Tuple`1[T1]
System.Tuple`2[T1,T2]
System.Tuple`3[T1,T2,T3]
...
```

Le résultat affiche des noms de classe sans leur espace de nom et postfixés par un chiffre, par exemple `1, qui représente le nombre d'argument de type qu'une classe générique attend lors de sa création, ce chiffre est aussi appelé l'arité.

On ne recherche que les classes génériques publiques qui ne sont pas imbriquées. Notez que l'assembly de cette classe est chargée par défaut :

```
[System.Collections.Generic.List[int]].Assembly.Location
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscorlib.dll
```

3 Nom de type d'une classe générique

L'usage de classe générique oblige, il me semble, à se pencher sur la manipulation d'un nom de classe générique.

Par abus de langage, on considère les termes *type* et *classe* comme équivalent.
Dans notre contexte, je dirais que l'on déclare des classes dans un code source et que l'on manipule des types à l'exécution.

3.1 Différence entre Type et RuntimeType

La classe *[Type]* permet de récupérer des informations sur un objet (des métadonnées), par exemple le nom de sa classe et l'espace de nom dans lequel elle est déclarée :

```
 #Identique à [System.Int32] $I=10
[int]$I=10
$I.Gettype().Fullname
System.Int32
$I.pstypeNames
System.Int32
System.ValueType
System.Object
```

Si on demande le type de l'objet renvoyé par la méthode *GetType()* :

```
$TypeDeLaVariableI=$I.Gettype()
$TypeDeLaVariableI.Gettype().Fullname
System.RuntimeType
```

On obtient *System.RuntimeType* qui est donc la classe définissant les métadonnées de notre objet.

Dans ce cas la propriété *PSTypeNames* proposée par PowerShell référence le système de réflexion :

```
$TypeDeLaVariableI.pstypeNames
System.RuntimeType
System.Reflection.TypeInfo
System.Type
System.Reflection.MemberInfo
System.Object
```

3.2 Créer un RuntimeType

Ceci crée le RuntimeType ***[System.int32]*** à partir de son nom de classe :

```
$IntegerType=[Type]'System.int32'
#ou
$IntegerType='System.int32' -as [Type]
$IntegerType -eq [System.int32]
True
```

Il est également possible d'utiliser une API de PowerShell :

```
[System.Management.Automation.LanguagePrimitives]::ConvertTo('System.in
t32',[Type])
```

Notez que ceci ne fonctionne pas :

```
$vartype=[Double]
[$vartype] $i=10
$vartype $i=10
```

Dans ce cas procédez ainsi :

```
$i=10 -as $vartype
$i.gettype().fullname
System.Double
```

Ceci est également possible :

```
$i=10 -as 'Double' # implicitement 'System.Double'
```

3.3 Créer une instance

On peut créer une instance à partir de la variable ***$IntegerType*** créée précédemment :

```
$IntegerType::New()
0
```

Note : la valeur zéro est la valeur par défaut du type.

Ou directement à partir d'un nom de classe :

```
[Activator]::CreateInstance('System.int32')
0
```

Cette dernière méthode permet également de créer des objets COM à partir d'un *classID* :

```
[GUID]$ClassID='{DCB00C01-570F-4A9B-8D69-199FDBA5723B}'
$TypeCOM=[Type]::GetTypeFromCLSID($ClassID)
$networkListManager = [Activator]::CreateInstance($TypeCOM);
$networkListManager.GetNetworkConnections()
IsConnectedToInternet IsConnected
--------------------- -----------
                 True        True
```

Donc, si l'on souhaite manipuler des types *via* une variable on peut procéder ainsi :

```
$X=[Type]'System.int32'
$X::new()
0
$X=[Type]'System.Double'
$X::new()
0
```

Sous réserve que la classe propose un constructeur sans paramètre ou de même signature :

```
$X=[Type]'System.String'
$X::new()
Surcharge introuvable pour « new » et le nombre d'arguments « 0 ».
$X::new('Texte')
Texte
```

Sachez que la classe *System.RuntimeType* est privé, seul le runtime du Framework DotNet peut créer des instances de cette classe.

L'usage de **New-Object** reste possible :

```
$L= new-object 'System.Collections.Generic.List[int]'
 #Enumére le contenant, pas le contenu
,$L|Get-Member
```

```
   TypeName : System.Collections.Generic.List`1[[System.Int32,
mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]]
…
```

3.4 Affichage d'un nom de type

La documentation de la classe *Type* propose l'exemple suivant :

```
Function ShowTypeInfo{
 param([Type] $T)
  Write-host "Name: $($T.Name)"
  Write-host "ToString Powershell: $T"
  Write-host "ToString dotNet: $($T.ToString())"
  Write-host "Full Name: $($T.FullName)"
  Write-host "Assembly Qualified Name: $($T.AssemblyQualifiedName)"
  Write-host
}
```

On utilise le nom du type sans les crochets, leur présence étant inutile:

```
ShowTypeInfo [System.String]
```

```
ShowTypeInfo : Impossible de traiter la transformation d'argument sur
le paramètre «T».
Impossible de convertir la valeur «[string]» du type «System.String» en
type «System.Type».
```

Pour cet exemple :

```
ShowTypeInfo System.String
```

```
Name: String
ToString Powershell: string
ToString dotNet: System.String


Full Name:  System.String


Assembly Qualified Name:  System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089
```

Rien de particulier, hormis la propriété AssemblyQualifiedName.

Notez que la transformation *"ToString: $($T)"* et *"ToString: $($T.ToString())"* diffère, la première utilise une API PowerShell lors de la substitution, la seconde une méthode native :

```
[Microsoft.Powershell.ToStringCodeMethods]::Type([System.String])
```

N'hésitez pas à tester ces deux formes.

3.4.1 Type générique fermé

« Pour une classe générique Node<T>, le code client peut référencer la classe en spécifiant un argument de type, pour créer un type construit fermé (Node<int>). »

Maintenant affichons un type générique fermé (on précise un type chargé en mémoire) :

```
ShowTypeInfo 'List[int]'
```

On obtient la même erreur que précédemment car le nom de type n'est pas complet, on doit soit le préciser :

```
ShowTypeInfo 'System.Collections.Generic.List[int]'
```

soit utiliser une facilité d'écriture de PowerShell 5.1 :

```
using namespace System.Collections.Generic
ShowTypeInfo 'List[int]'
```

L'usage de la clause ***using*** force PowerShell à rechercher dans les espaces de nom précisés lorsqu'il échoue à trouver un nom de type incomplet.

Ce qui nous affiche :

```
Name: List`1
ToString Powershell: System.Collections.Generic.List[int]
ToString dotNet: System.Collections.Generic.List`1[System.Int32]


Full Name:  System.Collections.Generic.List`1[[System.Int32, mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]]


Assembly Qualified Name:
System.Collections.Generic.List`1[[System.Int32, mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]],
mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

Ce résultat met en évidence les différents noms de type que l'on peut obtenir pour un type générique.

3.4.1.1 Le champ Name

Il affiche le nom de la classe sans l'espace de nom, mais postfixé par `1.
Ce chiffre représente le nombre d'argument de type que la classe attend lors de sa création, ce chiffre est aussi appelé l'arité.

Nombre que l'on a aperçu lors de la recherche des noms de classe générique dans un assembly :

```
System.Tuple`1[T1]         #Déclare UN argument de type
System.Tuple`2[T1,T2]      #Déclare DEUX arguments de type
System.Tuple`3[T1,T2,T3]   #Déclare TROIS arguments de type
```

PowerShell renvoi le même nom mais sans l'arité et sans l'espace de nom complet.

3.4.1.2 Le champs ToString

Il affiche le nom de la classe complet et le nom du type qui paramètre la classe générique fermée.

Le nom récupéré par Powershell peut servir à générer un nom de classe :

```
using namespace System.Collections.Generic
"[$([Type]'System.Collections.Generic.List[int]')]"
[System.Collections.Generic.List[int]]
```

La chaîne renvoyée est un nom de classe générique valide :

```
[System.Collections.Generic.List[Int]]

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     List`1                                   System.Object
```

3.4.2 Type générique ouvert

« Pour une classe générique Node<T>, le code client peut référencer la classe en laissant le paramètre de type non spécifié (Node<>). »

Avant d'afficher son nom essayons de construire le type :

```
[Type]'List[]'
Impossible de convertir la valeur « List[] » du type « System.String »
en type « System.Type ».
[Type]'System.Collections.Generic.List[]'
Impossible de convertir la valeur «System.Collections.Generic.List[]»
du type «System.String» en type «System.Type».
```

Ces deux constructions échouent. Poursuivons, ça tombera bien en marche :

```
using namespace System.Collections.Generic
[Type]'List`1[]'

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     List`1[]                                 System.Array
```

On construit ici un tableau de type ouvert, on peut créer le type mais pas l'instancier :

```
[Array]::CreateInstance([Type]'List`1[]',10)
Exception lors de l'appel de «CreateInstance» avec «2» argument(s):
«Impossible de créer des tableaux de type open. »
```

Seule l'instruction suivante construit un type ouvert :

```
[Type]'List`1'

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     List`1                                   System.Object
```

Donc pour construire un type générique ouvert on doit utiliser un nom de classe et préciser le nombre d'argument de type :

```
    $ClassName='System.Collections.Generic.List'
    $CountOfArguments= 1
    $OpenType=[Type]"$ClassName``$CountOfArguments"
```

On peut retrouver le type ouvert en utilisant la méthode *GetGenericTypeDefinition()* sur un type fermé :

```
$T=[System.Collections.Generic.List[int]].GetGenericTypeDefinition()
$T.Fullname
System.Collections.Generic.List`1
```

Maintenant affichons le type générique ouvert (on ne précise pas d'argument de type) :

```
using namespace System.Collections.Generic
ShowTypeInfo 'List`1'
Name: List`1
ToString Powershell: System.Collections.Generic.List`1[T]
ToString dotNet: System.Collections.Generic.List`1[T]


Full Name:  System.Collections.Generic.List`1
Assembly Qualified Name:  System.Collections.Generic.List`1, mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

3.4.2.1 Le champ Name

Son contenu ne diffère pas de celui affiché par un type générique fermé.
On peut construire un type générique ouvert si une clause *using*, pointant sur son espace de nom, a été exécutée au préalable.

3.4.2.2 Le champ ToString

En l'état on peut le réutiliser pour documenter, il précise le nombre d'argument de type et '**T**'qui est le nom du paramètre**.** Les deux appels renvoient un résultat identique.

3.4.2.3 Le champ FullName

Il peut servir à construire un type générique ouvert qui à son tour permettra de construire un type générique fermé :

```
$OpenGenericType = [Type]'System.Collections.Generic.List`1'
$closedType = $OpenGenericType.MakeGenericType([Int])
 # Ou
 # $Params=[int]
 # $closedType = $OpenGenericType.MakeGenericType($Params)

$closedType
IsPublic IsSerial Name           BaseType
-------- -------- ----           --------
True     True     List`1         System.Object
$closedType.tostring()
System.Collections.Generic.List`1[System.Int32]
```

La méthode ***MakeGenericType*** crée un type fermé à partir duquel on peut créer une instance :

```
$List=$closedType::New()
$List.add(1)
$List
1
```

On présente souvent une classe comme un modèle d'objet permettant de créer une multitude d'instances de classe (des objets), une classe générique (ouverte) permet de créer une multitude de types (des classes), on peut dire qu'une classe générique est un modèle de type.

3.4.3 Le champ AssemblyQualifiedName

Il permet de connaitre le nom complet d'un type afin de charger son assembly s'il n'est pas chargé.

4 Typer les paramètres

En PowerShell, la construction ***@( )*** crée par défaut un tableau d'objets, ce qui est incompatible avec, par exemple, une liste générique d'entier :

```
$L= new-object 'System.Collections.Generic.List[Int]'
$L.AddRange(@(1..20))
```

```
Impossible de convertir l'argument «collection» (valeur
«System.Object[]») de «AddRange» en type
«System.Collections.Generic.IEnumerable`1[System.Int32]»
```

Il est nécessaire dans ce cas de typer le tableau avec le même type que celui de la classe générique utilisée :

```
$L.AddRange(([Int[]]@(1..20)))
```

Par défaut la construction *@()* crée une instance d'un tableau contenant des éléments de type *object*, mais le principe d'une collection générique est qu'elle ne peut contenir que des éléments du type déclarée, ici [Int].

Lors de l'initialisation de cette classe générique, l'usage de la construction suivante reste possible :

```
$L=[System.Collections.Generic.List[int]] @(42,10,11)
$L
```

Sous réserve que la conversion réussisse, ici on utilise un cast au lieu d'un appel de constructeur. Ce cast transforme un tableau d'objet, ne contenant que des entiers, en un tableau d'entier puis en une collection générique d'entiers.

4.1.1 Opérateur +=

Sur une collection l'usage de l'opérateur += modifie le type de la collection, car il crée un nouveau tableau d'objet afin d'y ajouter un nouvel élément :

```
$L +=150
$L.GetType()
```

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

Une bonne raison de ne plus utiliser cette construction et préférer les méthodes de la classe :

```
$L.Add(150)
```

Notez qu'ici la méthode *Add()* ne renvoi aucune valeur, à la différence d'un ***ArrayList*** par exemple.

5 Délégué générique

Certaines classes génériques définissent des méthodes utilisant en paramètre des délégués. En simplifiant, sous DotNet un délégué est similaire à un pointeur de méthode, c'est-à-dire que l'on passe en paramètre l'adresse d'une méthode.

En PowerShell l'usage d'un script block est similaire :

```
Dir c:\temp\t*.txt |
 Where {$_.CreationTime.Date -eq [system.DateTime]::Today.Date}
```

Instruction qui peut être réécrite en paramétrant le cmdlet *Where-object* :

```
$FiltreAujourdhui= {$_.CreationTime.Date -eq [system.DateTime]::Today.Date}

Dir c:\temp\t*.txt | Where $FiltreAujourdhui

$FiltreHier={$_.CreationTime.Date -eq
[system.DateTime]::Today.AddDays(-1).Date}

Dir c:\temp\t*.txt | Where $FiltreHier
```

On paramètre donc une méthode en lui passant en argument du code.
Voir cet exemple basé sur des fonctions.

Un délégué générique permet de paramétrer le type de ses arguments :

```
public delegate void Action<in T>(T obj);
```

On s'assure ainsi, lors d'un appel de méthode utilisant un délégué, de manipuler le même type que celui de la classe générique :

List<T>.ForEach(Action<T>)

5.1 A propos de l'inférence de type

Comme pour la manipulation d'un nom de classe générique, on doit aborder cette notion avant de continuer.

Selon Wikipédia,
*« L'**inférence de types** est un mécanisme qui permet à un compilateur ou un interpréteur de rechercher automatiquement les types associés à des expressions, sans qu'ils soient indiqués explicitement dans le code source. »*

Dans l'exemple suivant on surcharge une méthode, c'est à dire qu'elle porte le même nom mais le type de ses paramètres diffère :

```
Add-Type @'
public static class BasicTest
{
  public static string UneMethode(string S, int end){
   return "UneMethode(string S, int end)";
  }
  public static string UneMethode(string S, string end){
   return "UneMethode(string S, string end)";
  }
}
'@
```

Le parseur PowerShell va tenter de **déduire** quelle méthode appeler selon le type du second argument :

```
[int]$o=10
[BasicTest]::UneMethode('s',$o)
UneMethode(string S, int end)
[String]$o='s'
[BasicTest]::UneMethode('s',$o)
UneMethode(string S, string end)
```

Si l'on souhaite appeler la seconde surcharge avec en paramètre un entier on doit préciser son type :

```
[BasicTest]::UneMethode('s',[string]10)
UneMethode(this string S, string end)
```

Dans ce cas on annule le mécanisme d'inférence car on précise notre intention.

Voir également : Inférence de type

5.2 Reconnaitre l'usage d'un délégué

Le code suivant passe en paramètre un nom de fonction :

```
function un { write-host 'F° un'}

function deux { write-host 'F° deux'}

function Invoke-FonctionName {
    param($Name)
    &$Name
}
Invoke-FonctionName deux
Invoke-FonctionName un
```

Et c'est l'opérateur **&** dans le corps de la fonction, nommée *Invoke-FonctionName*, qui se charge de la résolution de l'appel. Ce code ne reçoit pas de délégué.

Si on trace l'appel de l'opérateur **&**:

```
DÉBOGUER : CommandDiscovery Information: 0 : Looking up command: un
```

Idem pour l'instruction suivante, elle ne reçoit pas de délégué :

```
Invoke-Expression "write-host 'F° un'"
```

Ici on transforme du texte en un code puis on l'exécute. En interne on manipule bien une adresse mais c'est l'adresse d'une chaîne et c'est le cmdlet **Invoke-Expression** qui se charge de la transformation du texte en un code puis de son exécution.

L'exemple suivant :

```
Invoke-FonctionName {Write-Warning "Méthode anonyme"}
```

Reçoit une donnée de type scriptblock, l'usage de l'opérateur **&** reste valide.

Si on trace l'appel de l'opérateur **&**:

```
DÉBOGUER : CommandDiscovery Information: 0 : Looking up command: Write-
Warning
```

PowerShell 'exécute directement' le code. Ici on manipule donc un objet similaire à un délégué anonyme, on utilise l'adresse du paramètre qui pointe sur une déclaration de code.

Note : un scriptblock PowerShell doit être transformé (par le runtime) pour être considéré comme un délégué anonyme DotNet.

Si on parle de méthode anonyme on peut utiliser son contraire, à savoir une méthode nommée :

```
Invoke-FonctionName ${function:un}
```

Ici on ne passe plus un scritpblock, mais quelque chose qui pointe sur le code de la fonction nommée 'MaFonction' :

```
DÉBOGUER : PathResolution Information: 0 : Resolving MSH path
"function:un" to PROVIDER-INTERNAL path
```

Donc quelque chose similaire à la description d'un délégué.

La différence entre ces deux dernières constructions de code se visualise à l'aide de l'AST :

```
{Write-Warning "Méthode anonyme"}.ast
Attributes         : {}
UsingStatements    : {}
ParamBlock         :
BeginBlock         :
ProcessBlock       :
EndBlock           : Write-Warning "Méthode anomyme"
DynamicParamBlock  :
ScriptRequirements :
Extent             : {Write-Warning "Méthode anomyme"}
Parent             : {Write-Warning "Méthode anomyme"}
${function:un}.ast
IsFilter   : False
IsWorkflow : False
Name       : un
Parameters :
Body       : { write-host 'F° un'}
Extent     : function un { write-host 'F° un'}
Parent     : function un { write-host 'F° un'}
```

Dans les deux cas on reçoit bien l'adresse d'un objet qui pointe sur du code. C'est le Framework PowerShell qui se charge de rechercher la bonne adresse du code, nous n'avons pas à nous en occuper.

5.3 Retrouver les délégués disponibles

Précédemment le code utilisé pour retrouver les classes génériques ne renvoyait pas les types délégués, voici son adaptation :

```
[System.Collections.Generic.List[int]].Assembly.ExportedTypes|
Where-Object {
   $_.IsPublic -and $_.IsGenericType -and $_.IsInterface -eq $false -
and $_.IsNested -eq $false -and
$_.IsSubclassOf([System.MulticastDelegate])
 }|
 Foreach-Object {"$_"}
System.Action`1[T]
System.Action`2[T1,T2]
...
System.Func`1[TResult]
System.Func`2[T,TResult]
...
```

Note : Je suppose ici que les délégués utilisés par la classe générique se trouvent dans le même assembly.

5.3.1 System.Action : Action<T>

Le type System.Action *encapsule une méthode ayant un seul paramètre et ne retournant aucune valeur.*
Utilisons comme exemple la méthode *Foreach* de la classe générique ***List*** :

void ForEach(System.Action[T] action)

```
$names=[System.Collections.Generic.List[String]]::New()
$names.Add("Bruce")
$names.Add("Alfred")
$names.Add("Tim")
$names.Add("Richard")
```

Premier usage en précisant le type du délégué

```
$names.ForEach([System.Action[String]]{
  param ( [String]$Name)
    Write-host "$Name, action !"
})
Bruce, action !
…
```

Ou sans le préciser :

```
$names.ForEach({
  param ( [String] $Name)
    Write-host "$Name, action !"
})
Bruce, action !
…
```

Maintenant utilisons le code d'une fonction :

```
function Print{
  param([string] $s)
    Write-host "Print $s"
}
$names.ForEach(${function:Print})
```

Cela fonctionne également.

Si on modifie le type et le contenu de la liste on déclenche une exception :

```
$names=[System.Collections.Generic.List[String]]::New()
$names.Add("1");$names.Add("2");$names.Add("Tim")

$names.ForEach({
    param ( [Int] $Name)
    Write-host "$Name, action !"
})
1, action !
2, action !
Exception lors de l'appel de «ForEach» avec «1» argument(s):
«Impossible de convertir la valeur «Tim» en type «System.Int32».
Erreur: «Le format de la chaîne d'entrée est incorrect.»
```

Notez qu'elle se déclenche lors de l'itération, puisqu'en interne PowerShell convertit implicitement les données de la liste.

On constate que lors du parsing sous PowerShell aucun contrôle de type n'est fait. L'exemple suivant indique qu'**aucune inférence de type sur la clause *Param*() ne se fait lors d'un cast** :

```
function Print{
   param([System.Io.File] $s)
   Write-host "Print $s"
}
 (${function:Print} -as [System.Action[Int]]).ToString()
System.Action`1[System.Int32]
```

Un scriptblock peut donc être transformé en n'importe quel type de délégué :

```
(${function:Print} -as [System.Action[String]]) | Format-List
Method : Void lambda_method(System.Runtime.CompilerServices.Closure,
System.String)
Target : System.Runtime.CompilerServices.Closure
```

Note : les membres affichés ici correspondent à un objet interne de gestion d'un délégué codé en POWERSHELL et c'est lui qui 'porte' la signature du délégué.

```

{} -as [System.Action[String,int]] | Format-List
Method : Void lambda_method(System.Runtime.CompilerServices.Closure,
System.String, Int32)

{} -as [System.Func[String,Int]] | Format-List
Method : Int32 lambda_method(System.Runtime.CompilerServices.Closure,
System.String)
```

Note : le délégué Action propose des signatures avec 16 paramètres maximum.

En aucun cas nous ne sommes assurés que le code PowerShell utilisé respecte le type du délégué (sa signature) :

```
$sb=[System.Action[String]]{}
$sb.pstypenames
```

```
System.Action`1[[System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]
System.MulticastDelegate
System.Delegate
System.Object
```

En revanche le type du délégué générique doit correspondre au type de la classe générique :

```
$names.ForEach([System.Action[Int]]{
 param ( [String]$Name)
 Write-host "$Name, action !"
})
```

```
Impossible de convertir l'argument «action» (valeur
«System.Action`1[System.Int32]») de «ForEach» en type «
System.Action`1[System.String]»: ...
```

La responsabilité de ce contrôle incombe donc à ceux et celles qui codent.

Attention, la variable automatique ***$_*** n'est pas renseignée dans cet exemple :

```
$Numbers=[System.Collections.Generic.List[Int]]::New()
$Numbers.AddRange([int[]]@(1..5))
$Numbers.ForEach({
    Write-host "'$_', action !"
})
```

```
'', action !
...
```

Mais bien dans celui-ci :

```
&{
 Dir|
  Foreach-Object {
   $Numbers.ForEach({Write-host "'$_', action !"})
  }
}
```

5.3.2 System.Func : Funct<T,TResult>

Le type System.Func e*ncapsule une méthode qui a un seul paramètre* `(T)` *et qui retourne une valeur du type spécifié par le paramètre* `TResult`.

Pour cet exemple on utilisera la classe *[System.Linq.Enumerable]* qui propose de nombreuses méthodes, par exemple ***Select*** (Certains points technique seront abordés plus avant dans le texte).

C'est une méthode statique renvoyant un énumérateur générique sur une collection de type *TResult*

static System.Collections.Generic.IEnumerable[TResult]

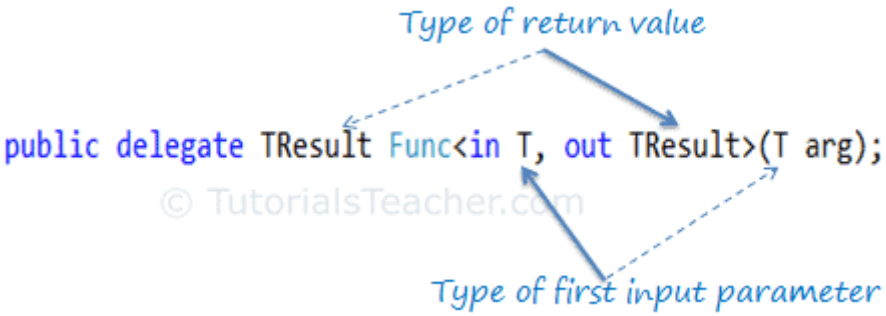
Elle attend une source de données en entrée et renvoi un résultat :

Select[TSource, TResult]

Son premier paramètre est un énumérateur générique, le second paramètre est un délégué de type ***System.Func*** *:*

System.Collections.Generic.IEnumerable[TSource] source,
System.Func[TSource,TResult] selector

Vous trouverez sur la page suivante (https://www.tutorialsteacher.com/csharp/csharp-func-delegate) divers graphiques détaillant le rôle des paramètres du délégué *System.Func :*



La définition complète :

static System.Collections.Generic.IEnumerable[TResult] Select[TSource, TResult](System.Collections.Generic.IEnumerable[TSource] source, System.Func[TSource,TResult] selector)

On constate que la relecture de certaines définitions de membre générique nécessite un peu plus d'effort, même si cette définition respecte cette règle.

Essayons avec un scriptblock :

```
[System.Linq.Enumerable]::Select($Names, { param($S) $S.Length})
Surcharge introuvable pour « Select » et le nombre d'arguments « 2 ».
```

Bien que le nombre de paramètre soit correct, ce message d'erreur indique qu'il ne trouve pas de surcharge pour cette méthode. C'est-à-dire une signature correspondant aux types des paramètres

C'est surtout le type du second paramètre qui implique l'échec de l'appel, en tout cas PowerShell ne peut l'utiliser en l'état, le contrôle des types nécessite ici d'être plus explicite que dans l'exemple vu précédemment dans le chapitre sur *Action<T>*.

Le champ **FullyQualifiedErrorId**,de l'objet *Error*, contient *MethodCountCouldNotFindBest* ce qui peut signifier que PowerShell est incapable d'inférer le type du scriptblock. Dans ce cas on précise le type :

```
$names=[System.Collections.Generic.List[String]]::New()
$names.Add("1");$names.Add("2");$names.Add("Tim")

$Delegate=[System.Func[String,Int]]{ param($S) $S.Length}
[System.Linq.Enumerable]::Select($Names, $Delegate)
1
1
3
```

On rencontre le même problème à l'exécution si on modifie le type du paramètre du scriptblock :

```
$Delegate=[System.Func[String,Int]]{ param([int]$S) $S.Length}
[System.Linq.Enumerable]::Select($Names, $Delegate)
1
1
Impossible de convertir la valeur «Tim» en type «System.Int32». Erreur:
«Le format de la chaîne d'entrée est incorrect.»
```

Un autre cas d'exception mais cette fois sur la valeur de retour, car le cmdlet *Get-ChildItem* ne renvoie pas d'objets de type entier :

```
$Delegate =[System.Func[String,Int]]{ param([int]$S) Get-ChildItem }
[System.Linq.Enumerable]::Select($Names, $Delegate)
Une erreur s'est produite lors de l'énumération par le biais d'une
collection : Impossible de convertir la valeur «System.Object[]» du
type «System.Object[]» en type «System.Int32».
```

5.3.3 System.Predicate : Predicate<T>

Le type System.Predicate *représente une méthode qui définit un ensemble de critères et détermine si l'objet spécifié répond à ces critères.*

Utilisons cette fois la méthode ***FindAll*** :

```
$Names.FindAll
OverloadDefinitions
-------------------
System.Collections.Generic.List[string]
FindAll(System.Predicate[string] match)
```

Le paramètre automatique *$args* permet une écriture sans la clause param :

```
$names=[System.Collections.Generic.List[String]]
@('Bruce','Alfred','Tim', 'Richard')


$Names.FindAll({$args[0].Contains('i')})
Tim
Richard
```

5.4 Exemple d'usage de délégué avec PSReadline

Le cmdlet *Set-PSReadlineOption* du module PSReadline déclare deux paramètres en tant que délégué générique fermé :

```
Get-Help Set-PSReadlineOption -Parameter CommandValidationHandler
-CommandValidationHandler <Action[CommandAst]>
  Specifies a ScriptBlock that is called from ValidateAndAcceptLine. If
an
  exception is thrown, validation fails and the error is reported.


Get-Help Set-PSReadlineOption -Parameter AddToHistoryHandler
-AddToHistoryHandler <Func[String, Object]>
  Specifies a ScriptBlock that can be used to control which commands
get
  added to PSReadLine history, and whether they should be saved to the
  history file.
```

Ceci n'insère plus les lignes saisie dans l'historique :

```
Set-PSReadLineOption -AddToHistoryHandler {param($line) return $false }
```

La visualisation via *Trace-Command* confirme la conversion du scriptblock :

```
DÉBOGUER : ParameterBinding Information: 0
 Trying to convert argument value from
System.Management.Automation.ScriptBlock
 to System.Func`2[System.String,System.Object]
```

Et ceci n'enregistre plus sur disque :

```
Set-PSReadLineOption -AddToHistoryHandler { param($line)
return [Microsoft.Powershell.AddToHistoryOption]::MemoryOnly }
```

Le type de retour est ***Object*** afin de traiter différents cas. Selon le code source une valeur booléen est identique à *[Microsoft.Powershell.AddToHistoryOption]::SkipAdding* .

5.5 Exemples avancés

L'exemple suivant est adapté d'un bug du projet Powershell sur GitHub. Il met en évidence une limite de Powershell lié à l'inférence de type.

5.5.1 System.Action avec une méthode d'instance avec Powershell 5.1

Un délégué étant une adresse, on peut pointer sur une méthode d'un objet :

```
using namespace System.Collections.Generic
$names=[List[String]] @('--','Tim','****','Tom')

Class MyClass {
  hidden [string] $s

  [void] Concat([string] $value) {
    $this.S += $value
    write-warning "s=$($this.s)"
  }
  [void] ConcatInt([Int] $value) {
    $this.S += $value
    write-warning "s=$($this.s)"
  }
}

$O=[myclass]::new()
$Names.ForEach($O.Concat)
Impossible de convertir l'argument «action» (valeur «void Concat(string
value)») de «ForEach» en type «System.Action`1[System.String]»: ...
```

Bien qu'on manipule une instance de classe, le message d'erreur signifie qu'il n'est pas possible de convertir un membre de type ***PSMethod*** en tant que délégué**.**

5.5.2 System.Action avec une méthode d'instance avec Powershell 7.0

Le code précédent fonctionne sous Powershell à partir de la version 7.0 :

```
$Names.ForEach($O.Concat)
WARNING: s=--
...
WARNING: s=--Tim****Tom
```

Avec la méthode *ConcatInt* l'erreur se déclenche avant l'itération :

```
$Names.ForEach($O.ConcatInt)
MethodException: Cannot convert argument "action", with value: "void
ConcatInt(int value)", for "ForEach" to type
"System.Action`1[System.String]":
```

Cette méthode attend en paramètre un type **Int**, là où *Foreach* attend un délégué de type **String**.

Voir également ce cas.

5.6 Les méthodes magique Foreach() et Where()

Pour faciliter certains traitements sur les collections Powershell, depuis la version 4.0, propose les méthodes *Foreach()* et *Where()*.

Elles autorisent l'utilisation d'un scriptblock :

ForEach(scriptblock expression) Where (scriptblock expression)

A mon avis ceci retarde l'usage d'un délégué et des génériques, vous trouverez des informations détaillées dans cet article.

6 Méthode générique

Une classe peut définir une méthode générique, ce qui permet de paramétrer un traitement au niveau d'une méthode uniquement.

Comme pour la déclaration d'une classe générique, celle d'une méthode générique précise un argument de type :

```
$code=@'
using System;
public class GenericList<T>
{
    public void Method(T value) { }
    public void SampleMethod<U>(U value) { }
}
'@
Add-Type $code
```

Une méthode est générique uniquement si elle possède sa propre liste de paramètres de type :

```
$O=[GenericList[int]]::new()
$O.GetType().GetMethods()|
 Where-Object {$_.name -match 'Method'}|
 Group-Object isGenericMethod
```

```
Count Name                      Group
----- ----                      -----
    1 False                     {Void Method(Int32)}
    1 True                      {Void SampleMethod[U]()}
```

Ici, seule la méthode nommée *SampleMethod* est générique.

Notez que nommer différemment les arguments de type permet de manipuler, dans le corps de la méthode, le type paramétrant la classe et le type paramétrant la méthode générique.

L'exemple suivant :

```
$code=@'
using System;
public class GenericList<T>
{
    public void Method(T value) { }
```

```
    public void SampleMethod2<T>(T value) { }
}
'@
Add-Type $code
```

Déclenche une erreur :

```
Add-Type : Avertissement en tant qu'erreur:
Le paramètre de type 'T' porte le même nom que le paramètre de type du
type externe 'GenericList<T>'
```

Prenons pour exemple la méthode générique ***FindAll*** de la classe *System.Array* :

```
 [System.Array].GetMembers()|
  Where-Object {
   $_.MemberType -eq 'Method' -and
   $_.Name -eq 'FindAll' -and
   $_.IsGenericMethod -and $_.IsStatic
  }|
  Foreach-Object{
     "$_"
  }
T[] FindAll[T](T[], System.Predicate`1[T])
[int[]]$T=1..100
[System.Array]::FindAll($T,[System.Predicate[int]]{($args[0] % 10) -eq
0})
```

On peut utiliser ***$Args[0]*** en lieu et place d'une clause *Param()*.

Powershell ne propose pas de syntaxe pour appeler de telle méthode :

T[] FindAll[T](T[], System.Predicate`1[T])

On doit se tourner vers les méthodes du système de réflexion pour les utiliser.

6.1 Inférence

Ici aussi c'est le type du paramètre qui détermine le type géré par la méthode générique.

Prenons la classe générique suivante qui déclare une méthode *WriteInformation()* et une méthode générique *WriteMethodInformation<U>* :

```
Add-Type @'
using System;
namespace Test {
 public class MaClasseGenerique<T>
 {
    T Data;

    public MaClasseGenerique(T Parametre)
    {
        this.Data = Parametre;
    }

    public void WriteInformation()
    {
        string S = typeof(T).FullName;
        Console.WriteLine(String.Format("type de l'instance = {0}", S));
    }
    public void WriteMethodInformation<U>()
    {
        WriteInformation();
        string S = typeof(U).FullName;
        Console.WriteLine(String.Format("type de la méthode sans
argument = {0}", S));
    }

    public void WriteMethodInformation<U>(U Donnée)
    {
        WriteInformation();
        string S = Donnée.GetType().FullName;
        Console.WriteLine(String.Format("type de la méthode = {0}",
S));
    }
 }
}
'@
```

La méthode qui n'est pas générique affiche le type du paramètre de la classe :

```
$c = [Test.MaClasseGenerique[int]]::new(10)
$c.WriteInformation()
type de l'instance = System.Int32
```

La méthode générique affiche le type du paramètre de la classe puis le type du paramètre de la méthode :

```
$c.WriteMethodInformation(10.2)
type de l'instance = System.Int32
type de la méthode = System.Double

$c.WriteMethodInformation('Test')
type de l'instance = System.Int32
type de la méthode = System.String
```

La recopie d'écran suivante, retravaillée, provient de l'éditeur Visual Studio. Elle affiche le type en cours lors de l'appel de la méthode *WriteMethodInformation<U>(U Donnée)*

```
class Program
{
    0 références
    static void Main(string[] args)
    {
        MaClasseGenerique<int> c = new MaClasseGenerique<int>(10);
        c.WriteInformation();
        Console.WriteLine();
        c.WriteMethodInformation<String>("test");
            void MaClasseGenerique<int>.WriteMethodInformation<string>(string Donnée)
        Console.WriteLine();
        c.WriteMethodInformation(10.2);
            void MaClasseGenerique<int>.WriteMethodInformation<double>(double Donnée)
        Console.WriteLine("return pour continuer.");
        Console.ReadLine();
    }
}
```

Le premier appel précise un type sur la méthode, le second est déduit du type de l'argument.

Si l'inférence se fait à l'aide du type d'un paramètre, l'usage d'une méthode générique sans paramètre va poser quelque problème. A la différence du C#, PowerShell ne propose pas de syntaxe d'appel qui préciserais le type du paramètre :

```
$c.WriteMethodInformation<String>()
```

A noter sur ce sujet cette proposition d'évolution pour Powershell version 7.

Note

Les mêmes règles d'inférence de type s'appliquent aux méthodes statiques et aux méthodes d'instance. Le compilateur peut déduire les paramètres de type d'après les arguments de méthode que vous passez. Il ne peut pas déduire les paramètres de type uniquement à partir d'une valeur de contrainte ou de retour. Par conséquent, l'inférence de type ne fonctionne pas avec les méthodes qui n'ont pas de paramètres.

6.2 MakeGenericMethod

Pour appeler sous Powershell une méthode générique sans paramètre on doit utiliser le système de réflexion de DotNet. Cette méthode couvre le cas où le paramètre de type est obtenu dynamiquement au moment de l'exécution.

Dans l'exemple suivant on appelle la méthode ***WriteMethodInformation<U>()*** :

```
$Instance = [Test.MaClasseGenerique[int]]::new(10)

[Type[]] $ParametersTypes=@()
$ClassType=$Instance.GetType()

$Method = $ClassType.GetMethod('WriteMethodInformation',$ParametersTypes)
$closedMethod = $Method.MakeGenericMethod([String])
$closedMethod.Invoke($Instance, @())        #Pas de valeur de retour
type de l'instance = System.Int32
type de la méthode sans argument = System.String
```

Cette méthode n'ayant pas de paramètre, la variable ***$ParametersTypes*** contient un tableau vide.

L'appel à ***MakeGenericMethod()*** construit notre méthode générique en la paramétrant avec un type, ici on choisit le type ***[String]***.

Enfin lors de l'invocation on passe l'objet concerné et le tableau d'argument, vide-lui aussi.

Pour une méthode statique on passe $*null* en lieu et place de l'objet ciblé (ici ***$Instance***), la méthode construite pointant déjà sur la classe ce paramètre est ignorée :

```
$closedMethod.Invoke($Null, @())
```

Si on veut reprendre cette approche pour appeler ***WriteMethodInformation<U>(U Donnée)*** ce n'est pas possible avec le framework 4.5. L'appel à *GetMethod()* dans le code précédant fonctionne car la première surcharge trouvée correspond à notre méthode générique sans paramètre.

Il est donc nécessaire de sélectionner notre méthode selon le nombre et le type des paramètres :

```
Foreach ($method in $ClassType.GetMethods()) {
  if ($Method.name -eq 'WriteMethodInformation') {
    $P=$Method.GetParameters()
    write-warning "count=$($P.Count)  $($Method.ToString())"
  }
}
AVERTISSEMENT : count=0  Void WriteMethodInformation[U]()
AVERTISSEMENT : count=1  Void WriteMethodInformation[U](U)
```

Le module PSGenericMethods propose un cmdlet pour simplifier ces appels :

```
$C = [Test.MaClasseGenerique[int]]::new(10)
Import-Module GenericMethods
$C | Invoke-GenericMethod -MethodName WriteMethodInformation -
GenericType string -ArgumentList 'Test'
type de l'instance = System.Int32
type de la méthode = System.String
$C | Invoke-GenericMethod -MethodName WriteMethodInformation -
GenericType string
type de l'instance = System.Int32
type de la méthode sans argument = System.String
```

Le Cmdlet *Invoke-GenericMethod* permet également l'appel d'une méthode générique statique.

Dans les exemples proposés précédemment on construit la méthode à chaque appel, la mise en cache de l'objet méthode améliorerait les performances.

Vous pouvez également consulter un des premiers article, sur le sujet.

6.2.1 La méthode Invoke() et l'usage de la virgule (comma operator)

Certaines signatures de méthode générique nécessitent une construction d'appel particulière.

Prenons la méthode statique *OfType* de la classe ***System.Linq.Enumerable*** qui filtre les éléments d'une collection en fonction du type spécifié :

```
public static System.Collections.Generic.IEnumerable<TResult>
 OfType<TResult> (this System.Collections.IEnumerable source)
```

On l'utilisera ainsi :

```
$Collection=@(1,2,'Test',10.5,8)
$method = [System.Linq.Enumerable].GetMethod('OfType')
#la méthode générique utilisée ne propose qu'une seule signature
$m = $method.MakeGenericMethod([int])
$m.Invoke($null, (,$Collection))
```

La signature de la méthode *Invoke()* est la suivante :

```
System.Object Invoke(System.Object obj, System.Object[] parameters)
```

Comme ici son appel se fait sur une méthode statique on passe la valeur ***$Null*** pour le premier paramètre.

Le second paramètre doit être précédé par une virgule afin de passer un tableau de paramètres contenant un tableau, car la signature de la méthode *OfType()* attend une collection d'objet :

```
OfType[TResult](System.Collections.IEnumerable source)
```

La méthode *Invoke()* doit pouvoir gérer toutes les signatures de méthode qui peuvent avoir un nombre différent de paramètres, on doit donc insérer dans le tableau du second paramètre un tableau.

Si on ne procède pas de cette manière chaque élément de la variable *$Collection* sera considéré comme un paramètre de la méthode. La méthode ***OfType*** ne proposant pas de signature ayant 5 paramètres, l'appel échoue :

```
$m.Invoke($null, $Collection)
Exception lors de l'appel de « Invoke » avec « 2 » argument(s) :
« Nombre de paramètres incorrects. »
```

Le message d'erreur ne précise pas que c'est l'appel interne de la méthode *OfType()* qui déclenche l'erreur.

Comportement en passant un tableau d'entier :

```
($Collection).GetType().Name
Object[]
($Collection)[0].GetType().Name
Int32
($Collection).Count
5
```

Comportement en passant un tableau de tableau :

```
(,$Collection)[0].GetType().Name
Object[]
(,$Collection)[0].GetType().Name
Object[]
(,$Collection).Count
1
```

Un autre exemple issu de ce blog :

```
$bytes = 1..6
$Stream = New-Object System.IO.MemoryStream $bytes
New-Object : Surcharge introuvable pour « MemoryStream » et le nombre
d'arguments « 6 ».
```

7 Classe générique contrainte

Une classe générique peut contraindre le ou les paramètres de type sur lesquelles elle peut intervenir. Ceci pour éviter des erreurs de compilation sur des opérations impossible avec certains types.

L'exemple suivant contraint la classe *GenericList<T>* où T doit être un type dérivé de la classe *Employee* :

```
Add-Type @'
using System;
namespace Test {
 public class Employee
 {
    public Employee(string name)
    { Name=name; }
    public string Name { get; set; }
 }
 public class GenericList<T> where T : Employee
 {
    public string GetName(T personne) { return personne.Name;}
 }
}
'@
```

Si on respecte la contrainte, le type demandé est accessible :

```
[Test.GenericList[Test.Employee]]

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    GenericList`1                            System.Object
```

Dans le cas où l'on tente d'utiliser une classe qui enfreint la contrainte, cela déclenche une exception. Mais sous Powershell le message d'erreur est difficile à interpréter :

```
[Test.GenericList[String]]
Type [Test.GenericList] introuvable...
```

La cause réelle n'est pas précisée, en C# on obtient l'erreur suivante (CS0311) :

Impossible d'utiliser le type 'string' comme paramètre de type 'T'dans le type ou la méthode générique 'GenericList<T>'. Il n'y a pas de conversion de référence implicite de 'string' en 'Test.Employee'.

Pour analyser la cause on peut soit lire le code source de la classe soit obtenir un message plus explicite :

```
$genericType = [Type]'Test.GenericList`1'
$closedType = $genericType.MakeGenericType([Int])
Exception lors de l'appel de «MakeGenericType» avec «1» argument(s):
«GenericArguments[0], 'System.Int32', sur 'Test.GenericList`1[T]'
ne respecte pas la contrainte de type 'T'.»
```

7.1 Méthode contrainte

Pour une méthode générique contrainte, le message d'erreur est également inadapté :

```
Add-Type @'
using System;
namespace Test {
 public class MaClasseGenerique<T>
 {
    public void WriteMethodInformation<U>(U Donnée) where U : class
    {
       string S = Donnée.GetType().FullName;
       Console.WriteLine(String.Format("type de la méthode = {0}", S));
    }
 }
}
'@
$c = [Test.MaClasseGenerique[int]]::new()
$c.WriteMethodInformation(10)
```

```
Surcharge introuvable pour « WriteMethodInformation » et le nombre
d'arguments « 1 ».
...
    + FullyQualifiedErrorId : MethodCountCouldNotFindBest
```

L'usage de la méthode *MakeGenericMethod()* permet d'obtenir un message un peu plus explicite :

```
$C|Invoke-GenericMethod -MethodName WriteMethodInformation -GenericType
int -ArgumentList 10
```

```
Exception lors de l'appel de «MakeGenericMethod» avec «1» argument(s):
GenericArguments[0], 'System.Int32',
sur 'Void WriteMethodInformation[U](U)' ne respecte pas la contrainte
de type 'U'.
```

Ainsi on sait qu'une contrainte est en cause, à vous de déterminer laquelle…

8 ETS et les génériques

La version 3 de Powershell ajoute le cmdlet **Update-TypeData** qui facilite la création d'extensions de type (ETS), avant cette version on utilisait uniquement un fichier .ps1xml.

Ce qui simplifie la création d'extension pour les types génériques :

```
$List_String = new-object 'System.Collections.Generic.List[String]'
$List_Int = new-object 'System.Collections.Generic.List[Int]'

,$List_String|gm -MemberType scriptmethod
,$List_Int|gm -MemberType scriptmethod

$TypeStringFullname=$List_String.GetType().FullName
Update-TypeData -TypeName $TypeStringFullname -MemberType ScriptMethod
-MemberName MyMethod -Value {
 Param( [Object[]] $ArgumentList )
  Write-host "MyMethod $ArgumentList"
}

,$List_String|gm -MemberType scriptmethod
,$List_Int|gm -MemberType scriptmethod
$List_String.MyMethod(@(1, ' test.'))
```

Ici seule la classe *Collections.Generic.List[String]* possédera la méthode nommée « MyMethod ».

Pour étendre cet ajout de membre à tous les types fermés d'une classe générique on doit forcer ETS à les reconnaitre. Lors de sa recherche sur les noms de type, ETS interroge le contenu de la propriété Powershell nommée *PSTypenames*,

C'est cette propriété qui est utilisée en interne pour déclencher l'ajout d'un membre pour une instance d'objet, on ne peut donc pas indiquer ceci :

```
Update-TypeData -TypeName System.Collections.Generic.List[*] …
```

Bartek Bielawski propose cette solution, si vous l'utilisez gardez à l'esprit qu'implicitement on suppose cette égalité :

```
$List_String = new-object 'System.Collections.Generic.List[String]'
$List_String.PSTypeNames[0] -eq $List_String.GetType().FullName
True
```

La présence du nom complet dans PSTypeNames semble être un bug d'implémentation.

Notez que dans le cas où on utilise un nom court, le cmdlet *Update-TypeData* transforme ce nom en un nom qualifié :

```
Get-TypeData -TypeName 'System.Collections.Generic.List*'|
 Remove-TypeData
Update-TypeData -TypeName 'Collections.Generic.List[String]' -
MemberType ScriptMethod -MemberName MyMethod -Value {
 Param( [Object[]] $ArgumentList )
  Write-host "MyMethod $ArgumentList"
}
Get-TypeData -TypeName 'System.Collections.Generic.List*'
TypeName
--------
System.Collections.Generic.List`1[[System.String, mscorlib, ...
```

Si on veut retrouver le nom court du type d'une instance (***System.Collections.Generic.List[String])*** il nous faut procéder comme ceci :

```
$Objet = new-object 'Collections.Generic.List[String]'
$TypeName="$($Objet.GetType())"
```

NOTE : on peut également utiliser l'API suivante pour récupérer le nom court du type :

```
[microsoft.powershell.ToStringCodeMethods]::Type($Objet.GetType())
```

8.1 Créer un raccourci de type avec un générique

On peut en créer un soit sur un type fermé :

```
$AcceleratorsType=
[PSObject].Assembly.GetType("System.Management.Automation.TypeAccelerators")
$AcceleratorsType::Add('ListString','System.Collections.Generic.List[String]')
$L=[ListString]::new();$L.GetType().FullName
System.Collections.Generic.List`1[[System.String, mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]]
```

Soit sur un type ouvert :

```
$AcceleratorsType::Remove('ListString')
$AcceleratorsType::Add('GenericList','System.Collections.Generic.List`1')


$L=[GenericList[Int]]::new();$L.GetType().FullName
System.Collections.Generic.List`1[[System.Int32, mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]]


$L=[GenericList[String]]::new();$L.GetType().FullName
System.Collections.Generic.List`1[[System.String, mscorlib,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]]
```

L'instruction ***'Using'*** est toutefois préférable à partir des versions Powershell version 5.0 :

```
using namespace System.Collections.Generic
```

A noter que la syntaxe suivante est prévue mais pas encore implémentée :

```
using type list = System.Collections.ArrayList
This syntax of the 'using' statement is not supported.
```

9 A propos de LINQ

LINQ permet d'interroger des collections à l'aide d'opérateurs pouvant être chaînés dans une requête unique (dans un langage compilé).

L'article suivant aborde dans le détail l'usage de LINQ avec Powershell. Son auteur indique au chapitre '**Generic LINQ Operators**' que seul 3 opérateurs sont génériques : *Cast*, *OfType*, et *Empty*.

Nous avons vue comment appeler une méthode générique statique sans paramètre ce qui est le cas de la méthode *Empty()*, nous avons également vu comment appeler une méthode d'extension générique, par exemple les méthodes *Cast()* et *OfType()*.

LINQ utilise les méthodes d'extension pour manipuler des collections de type différent à partir du moment où chaque type implémente l'interface ***IEnumerable<T>.***

De plus aucun opérateur LINQ ne modifie la collection d'origine, chacun renvoi un nouvel objet.

9.1 Méthode d'extension

Une méthode d'extension est une méthode statique qui définit en premier paramètre le type pour lequel elle s'applique (précédé du mot clé ***this*** en C#*)* suivi du nom des paramètres utilisé dans le corps de la méthode. Le mécanisme des méthodes d'extension permet de lier une ou des méthodes à un type sans modifier son code source.

Si on prend l'exemple suivant :

```
$source = @"
using System;
  public static class Test
  {
    public static DateTime EndOfTheMonth(this DateTime date)
    {
       var endOfTheMonth = new DateTime(date.Year, date.Month, 1)
                                .AddMonths(1)
                                .AddDays(-1);

      return endOfTheMonth;
   }
 }
"@
$Assembly=Add-Type -TypeDefinition $source -passthru
```

A la différence d'un appel en C# où l'enchaînement des appels se fait dans l'ordre :

```
Datetime.Now.EndOfTheMonth()
```

Sous Powershell c'est l'inverse, on doit utiliser la méthode statique, mais en précisant un objet ou une variable, de type *[DateTime]*, sur laquelle elle s'applique :

```
[Test]::EndOfTheMonth([Datetime]::now)
dimanche 31 janvier 2021 00:00:00
```

Un appel sans paramètre provoquera une exception.

Il est possible d'utiliser ce type de méthode d'extension via ETS, selon l'idée de Bart De Smet's. Ainsi on disposera d'une syntaxe d'appel similaire à celui du C# :

```
Import-Module ExtensionMethod
$Assembly| New-ExtendedTypeData -Path C:\temp\ -verbose
COMMENTAIRES : Type 'System.DateTime'
COMMENTAIRES : Method 'EndOfTheMonth'
COMMENTAIRES : Create the ETS file 'C:\temp\System.DateTime.ps1xml'
```

Le contenu du wrapper ETS généré :

```
Type 'C:\temp\System.DateTime.ps1xml'
<?xml version="1.0" encoding="utf-8"?>
<Types>
  <Type>
    <Name>System.DateTime</Name>
    <Members>
      <ScriptMethod>
        <Name>EndOfTheMonth</Name>
        <Script>
   switch ($args.Count) {
      # EndOfTheMonth([datetime] $date)
     0 { [Test]::EndOfTheMonth($this) }
     default {
    throw "No overload for 'EndOfTheMonth' takes the specified number
of parameters."
   }
    }
        </Script>
      </ScriptMethod>
    </Members>
  </Type>
</Types>
```

Reste à charger ce fichier de type pour accéder à la méthode :

```
Update-TypeData -PSPath 'C:\temp\System.DateTime.ps1xml'
[Datetime]::Now.EndOfTheMonth()
dimanche 31 janvier 2021 00:00:00
```

Dans ce contexte ***$This*** référence l'objet appelant cette méthode, s'il est du type attendu par la méthode d'extension, ici [DateTime], l'appel fonctionnera.

Notez qu'il est possible de déclarer des méthodes d'extension génériques.

Un module d'exemple à partir de DLL : https://github.com/LaurentDardenne/EtsDateTime

Ce site propose différentes méthodes d'extension : https://extensionmethod.net/

9.2 Interface

Une interface se focalise sur un comportement afin de l'associer à plusieurs hiérarchies de classes. Si on prend une voiture, un cheval et un nuage on va avoir du mal à factoriser un traitement commun aux trois classes. En revanche, si on se concentre sur la notion de déplacement, déclarer une interface identique sur ces trois classes est une solution pour implémenter notre traitement commun. On ne cherche plus à savoir ce qu'est l'objet mais ce qu'il sait faire (aptitude). Et c'est à la personne en charge du développement de préciser comment il le fait.

9.2.1 L'Interface IEnumerable

Cette interface**,** pour toutes les classes qui l'implémente, permet de parcourir une collection d'objet, un par un et du premier au dernier sans retour arrière.

A noter qu'une collection peut ne pas être ordonnée ni indexée, à moins d'ajouter une ou des interfaces autorisant ce comportement.

9.2.2 L'Interface IEnumerable<T>

Cette interface garantie qu'une classe est itérable et pourra être utilisées avec LINQ qui permet d'interroger des collections à l'aide d'opérateurs pouvant être chaînés dans une requête unique (dans un langage compilé).

Prenons l'exemple de la méthode *Reverse()*, sa signature précise en paramètre un IEnumerable générique et pas un type tableau :

```
"$([Linq.Enumerable].GetMethod('Reverse').GetParameters())"
System.Collections.Generic.IEnumerable`1[TSource] source
```

Les types tableau implémentent l'interface générique *IEnumerable<T>* :

```
[int[]]$T=1..2
$t.gettype().getinterfaces().fullname
...
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable`1[[System.Int32, …
```

Ceci est ajouté lors de la compilation par une mécanique interne à DotNet, ce qui permet à LINQ d'utiliser un tableau, bien qu'il ne soit pas générique :

```
[Linq.Enumerable]::Reverse($T)
2…
```

Notez qu'un type générique implémentant *IEnumerable<T>* renvoi ***$True*** dans les cas suivants :

```
$List=[System.Collections.Generic.List[int]]::New($T)
$List -is [System.Collections.Generic.IEnumerable[Int]]
#True
$List -is [System.Collections.IEnumerable]
#True
```

Car selon la déclaration de cette interface qui peut le plus peut le moins :

```
public interface IEnumerable<out T> : IEnumerable
```

9.3 Linq et les collections classiques

On doit pouvoir utiliser Linq avec les collections de DotNet 1.0 ou 2.0 qui ne sont pas génériques.

9.3.1 Array et IEnumerable<T>

Pour le type Array bien que l'interface *IEnumerable<T>* soit absente dans la déclaration de la classe, c'est le Framework DotNet qui ajoute son implémentation lors de l'exécution :

```
[int[]]$Numbers=1..10
$Numbers.GetType().GetInterfaces()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    IEnumerable
True     False    IList`1
True     False    IEnumerable`1 …
```

Ceci permet d'utiliser un objet de type *Array* avec Linq.

Voir aussi :
https://mattwarren.org/2017/05/08/Arrays-and-the-CLR-a-Very-Special-Relationship/

9.3.1 Propriété Count

Cette propriété se comporte différemment pour certaines classes :

```
[int[]]$Numbers=1..10
$CinqPremiersElement= [Linq.Enumerable]::Take($Numbers, 3)
$CinqPremiersElement.count
1
1
1
```

Dans ce cas on doit forcer l'énumération de la collection afin d'obtenir le nombre d'élément :

```
@($CinqPremiersElement).Count
3
```

Ou typer la variable recevant le résultat :

```
[Int[]]$CinqPremiersElement= [Linq.Enumerable]::Take($numbers, 3)
$CinqPremiersElement.count
3
```

Dans ce dernier exemple on manipule un type tableau et plus une collection *IEnumerable* générique.

9.3.2 DataTable

Toutes les classes ne sont pas concernées par cet ajout, par exemple le type Arraylist ou Datatable.

La fonction suivante recherche si un type implémente l'interface générique IEnumerable :

```
${Function:Test-GenericIEnumerable}=.{
    #On recherche le type ouvert de l'interface générique IEnumerable
    $GenericIEnumerable=[Type]'System.Collections.Generic.IEnumerable`1'

    Return {
      #La variable $InputObject implémente-t-elle l'interface générique IEnumerable ?
      param(
        [ValidateNotNull()]
        $InputObject
      )
        foreach ($Interface in $InputObject.GetType().GetInterfaces())
        {
           if ($Interface.IsGenericType)
           {
                 #On suppose une seule implémentation de l'interface
                 # générique IEnumerable (https://stackoverflow.com/a/7852650)
               if ($Interface.GetGenericTypeDefinition() -eq $GenericIEnumerable)
               { return $true }
           }
        }
        return $false
    }.GetNewClosure()
}
[void][Reflection.Assembly]::LoadWithPartialName("System.Data.DataSetExtensions");
$Dt = New-Object System.Data.DataTable
[void]$Dt.Columns.Add( 'C1')
[void]$Dt.Columns.Add( 'C2')
[void]$Dt.Rows.Add( '1','2')
[void]$Dt.Rows.Add( 'Y','Z')

$Dt -is [System.Collections.IEnumerable]
#false
$Dt.Rows -is [System.Collections.IEnumerable]
#True
Test-GenericIEnumerable $Dt.Rows
#False
$A = New-Object System.Collections.ArrayList
$A -is [System.Collections.IEnumerable]
#True
Test-GenericIEnumerable $A
#False
```

On a donc une collection itérable mais qui ne peut pas être utilisée directement avec Linq.

Pour utiliser un opérateur Linq avec un DataTable on doit transformer la collection :

```
$Enumerable=[System.Data.DataTableExtensions]::AsEnumerable($Dt);
```

Pour l'Arraylist on utilisera *MakeGenericMethod* sur l'opérateur [Linq.Enumerable]::Cast.

9.4 Exécution Différée

Reprenons l'exemple utilisé à propos de la gestion de la propriété count :

```
[int[]]$Numbers = 1..10
#Les 5 premiers
$C= [Linq.Enumerable]::Take($Numbers, 5)
```

L'affichage des détails du type de la collection précise plusieurs points :

```
,$C|Get-Member

   TypeName :
System.Linq.Enumerable+<TakeIterator>d__25`1[[System.Int32, mscorlib,
Version=4.0.0.0, Culture=neutral,PublicKeyToken=b77a5c561934e089]]


Name           MemberType   Definition
----           ----------   ----------
…
<>3__count     Property     int <>3__count {get;set;}
<>3__source    Property     System.Collections.Generic.IEnumerable[int]
<>3__source {get;set;}
…
```

Le type renvoyé est une classe spécifique à LINQ et l'objet propose deux propriétés particulières.

La propriété '*<>3__count*' contient le nombre d'objet renvoyés :

```
$c.'<>3__count'
5
```

Celui précisé lors de l'appel à la méthode *Take()* et la propriété '*<>3__source*' référence la collection **$numbers** :

```
$c.'<>3__source'
1..10
[System.Object]::ReferenceEquals($c.'<>3__source',$numbers)
True
```

Ceci est dû au fait que certaines opérations de LINQ sont implémentées à l'aide de l'exécution différée :

« La valeur de retour immédiate est un objet qui stocke toutes les informations nécessaires à l'exécution de l'action. La requête représentée par cette méthode n'est pas exécutée tant que l'objet n'est pas énuméré en appelant directement sa méthode GetEnumerator() ou en utilisant foreach. »

Notez que le nom de type diffère sous *Powershell Core,* de plus tous les opérateurs LINQ ne permettent pas l'exécution retardée, comme indiqué sur le tableau suivant (cf. colonne *Laziness*)

https://www.red-gate.com/simple-talk/dotnet/net-development/visual-lexicon-linq-wallchart/

Ce lien contient certains détails à propos du nommage de la classe suivante :

System.Linq.Enumerable+<TakeIterator>d__25

Pour de l'exploration de données dans la console on peut utiliser ceci :

```
[System.Collections.Generic.List[int]]$numbers = 1..10
$s=[Linq.Enumerable]::skip($numbers,3)
$s.'<>3__count'
#3
$s.'<>3__source'
#1..10
$s
#4..10
$numbers.AddRange([int[]]@(11..15))
$s
#4..15
```

Comme l'objet dans la variable $S mémorise une référence sur $Numbers, la demande d'affichage de $S dans la console appel de nouveau l'itérateur spécialisé crée par le premier appel à ***[Linq.Enumerable]::Skip***.

Bien que le contenu de la collection *$Numbers* ait changé, il n'est pas nécessaire ici d'exécuter à nouveau l'opérateur **Skip** (notez ici le type de la variable *$Numbers*).

Par contre si on crée une nouvelle collection, la variable ***$S*** à toujours une référence sur l'ancienne collection, elle n'est donc pas supprimée en mémoire :

```
$numbers = 30..40
$s
#4..15
```

Si on doit itérer de nombreuses fois sur le résultat, il est préférable de le transformer afin d'optimiser les accès :

```
$T=$S -as [Int[]]
$S=$null
```