

La notion d'objet sous PowerShell

Par Laurent Dardenne, le 23 mai 2009.



Niveau		
Débutant	Avancé	Confirmé
<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>

Dans les langages de programmation orientée objet (POO), les objets sont les éléments principaux avec lesquels on interagit, il en est de même avec PowerShell.

Ce shell est souvent présenté comme orienté objet alors que les premières personnes concernées par ce produit, à savoir les administrateurs systèmes, n'ont pour la plupart qu'une vague idée des principes de base de la programmation orientée objet. J'essaierais donc dans ce tutoriel de leur indiquer comment ces principes s'appliquent sous PowerShell.

Merci à shawn12 pour sa relecture et ses corrections.

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

Chapitres

1	COMMENT FAISONS-NOUS JUSQU'A MAINTENANT ?	3
2	BIENVENUE DANS LE GROUPE	5
2.1	COMMENT MANIPULER UN OBJET ?	6
3	QUELLE CLASSE !	9
3.1	CREATION D'OBJET MAIL	9
3.2	LA NOTION D'ASSEMBLY	11
4	AH MES AÏEUX !	12
4.1	MEMBRES STATIQUES	16
4.2	SURCHARGE DE METHODE	16
5	J'AI UNE DE CES FORMES !	17
5.1	JOLI CASTING !	19
5.1.1	<i>Comment tester si un objet est d'un type particulier ?</i>	20
6	QUI SORT LES POUBELLES ?	20
6.1	CREATION D'OBJET MAIL, SUITE ET FIN	21
6.2	À PROPOS DE LA VERSION 2 DE POWERSHELL	22
7	GET-MEMBER OU COMMENT DETAILLER UN OBJET	23
7.1	UN CADRE DYNAMIQUE	25
8	ET J'AI CRIE A L'AIDE	25
8.1	QUELQUES LIENS UTILES	31
9	CONCLUSION	31
10	LIENS	32

1 Comment faisons-nous jusqu'à maintenant ?

Depuis toujours Microsoft propose, que ce soit dans le shell de MS/DOS ou Windows, un langage de script communément nommé les batchs.

Ce langage comporte peu d'instructions et manipule des informations textuelles. Étudions comme tâche la récupération du nom des fichiers présents dans un répertoire.

Dans un premier temps, et à l'aide de la commande **DIR**, récupérons la liste des fichiers :

```
Dir c:\temp\*.tmp
```

Cette instruction renvoie de nombreuses lignes sous forme de chaînes de caractères :

```
Répertoire de c:\temp
07/03/2009 12:29 <REP>      ispC.tmp
07/03/2009 12:29 <REP>      SIT26534.tmp
08/03/2009 13:30          512 ~DF8058.tmp
...
08/03/2009 14:28      49 512 ~DFFACB.tmp
08/03/2009 14:28          512 ~DFFADF.tmp
          9 fichier(s)      411 784 octets
```

On doit donc analyser chaque ligne afin d'extraire le nom du fichier qu'elle contient, puis appeler un traitement en lui passant en paramètre ce nom de fichier :

```
for /f "tokens=4" %i in ('dir c:\temp\*.tmp^|find ".tmp"^|find /V
"<REP>"') do echo traitement de %i
```

Si plus tard on souhaite récupérer la date de création de chaque fichier, on ajoutera à l'analyse des chaînes de caractères, émises par la commande **DIR**, le traitement dédié à cette opération. Ici on manipule bien un concept de fichier, mais par l'intermédiaire d'une chaîne de caractères qui contient tout ou partie des informations associées à un fichier.

Premier point : l'accès aux informations propre à un fichier dépend d'une analyse de texte. Ce n'est pas tant l'analyse de texte qui pose problème ici, mais le fait que l'information ne soit disponible que dans un format textuel.

Au fil du temps, et afin de nous faciliter ce type d'analyse, l'équipe de développement de Microsoft en charge du Shell a ajouté de nouvelles fonctionnalités à la commande **DIR** ainsi qu'à l'instruction de boucle **For** :

```
Dir c:\temp\*.tmp /B
```

```
ispC.tmp
SIT26534.tmp
~DF8058.tmp
~DF924C.tmp
~DFAC10.tmp
~DFBBC7.tmp
~DFC9E6.tmp
~DFCAEF.tmp
~DFDF63.tmp
~DFFACB.tmp
~DFFADF.tmp
```

Mais pour récupérer l'information concernant la date de création de ces fichiers, cela n'est pas possible en l'état. Pourtant, ces informations sont spécifiques à chaque fichier ou répertoire.

On connaît le numéro de la colonne où débute le nom de fichier, celle où débute la date de création et ainsi de suite. Chaque information liée à un objet fichier est accessible soit par une analyse de chaîne de caractères soit directement, mais dans ce cas les autres informations ne sont plus accessibles. Ici dans une chaîne de caractères, d'un format ou d'un autre, les informations sont dispersées, mais restent structurées, malheureusement aucune instruction ne renvoie de chaînes de caractères contenant toutes les informations propres à un fichier.

Notre souci se situe non pas dans l'accès aux informations, car d'une manière ou d'une autre on accédera aux informations souhaitées, mais dans la spécialisation de l'accès. Le traitement d'analyse domine la donnée. Sous ce Shell, c'est-à-dire avec un fichier batch, on doit spécifier COMMENT récupérer les données qui nous intéressent.

```
rem GetDirName "c:\temp\t r\1234567890.txt" DirectoryName
rem Renvoie DirectoryName=c:\temp\t r\
for /F "tokens=*" %%I in ('echo %1') do set Result=%%~dpI
set %2=%Result%

-----

rem GetFileName.cmd "c:\t r\1234567890.txt" OnlyFileName
rem Renvoie : OnlyFileName=1234567890
for /F "tokens=*" %%I in ('echo %1') do set %2=%%~nI
```

Second point : l'accès aux informations est spécialisé et atomisé. Elles sont toutes accessibles, mais pas toutes au même moment ni de manière uniforme.

De plus, nous n'avons pas abordé les traitements spécifiques à un fichier d'après un ou plusieurs critères :

Afficher les 3 derniers fichiers .TMP créé ce jour ayant une taille supérieure à 10 ko.

Nous devons créer un traitement spécialisé pour chaque requête de ce type, mais surtout une analyse de texte pour chaque information à récupérer. Il est tout à fait possible, plus ou moins facilement, avec ou sans programme externe de réaliser ces traitements, mais vous imaginer bien qu'avec cette manière de procéder, la combinaison de critères ne va pas nous faciliter le développement.

Eh oui, le scripting c'est du développement qui, c'est vrai, ne nécessite pas de préciser autant de points que dans un langage dit évolué.

Si donc on énonce que dans notre cas un des problèmes est dû à une dispersion d'informations, propres à un fichier, essayons de manipuler sous PowerShell notre fichier comme un objet regroupant tout ce qui le concerne et rien d'autre.

Ce que fait déjà, en partie, la commande **DIR**, mais, elle, renvoie des espaces, des <REP>, des trucs et des machins, du style «*Répertoire de c:\temp*», «*9 fichier(s) 411 784 octets*».

Certaines informations qui n'ont pas grand-chose à voir avec un fichier, mais avec son propriétaire, de plus on mélange les données et leur présentation.

Troisième point : les informations sont dispersées et ne sont pas hiérarchisées, c'est le traitement d'accès aux données, lors de leur affichage, qui détermine la hiérarchie.

2 Bienvenue dans le groupe

Un objet peut être vu comme un conteneur regroupant à la fois des informations et des actions.

Les informations nous renseignent sur les caractéristiques propres à un objet, par exemple pour un fichier sa taille et sa date de création, pour un processus son occupation mémoire. Il est possible d'avoir plusieurs fichiers, dans ce cas chaque objet fichier possède des caractéristiques communes, mais dont le contenu pourra être différent.

C'est-à-dire que la date de création pourra être différente pour chaque fichier, mais tous les fichiers porteront une information nommée "date de création". Avec la commande DIR chaque ligne de texte regroupait déjà ces informations, à priori la notion d'objet n'apporte rien de nouveau de ce côté là.

La notion d'objet implique de classer les objets selon leurs caractéristiques propres, comme on range des fichiers dans des dossiers thématiques, on répertorie les objets ayant les mêmes caractéristiques dans une classe. Dans le premier cas des erreurs de classement restent possible, mais pas dans le second, un fichier n'est pas une imprimante et *vis et versa*.

Cette notion de classe nous permettra de distinguer les différents objets que l'on sera amené à manipuler. Comme dit précédemment le concept de fichier a peu à voir avec celui d'imprimante. Nous aurons donc des objets de la classe fichier et des objets de la classe imprimante.

Dès maintenant on peut voir que notre ligne de texte peut encore supporter la comparaison avec un objet, mais on sent que la différence se situe à un autre niveau. Une ligne de texte peut contenir n'importe quoi c'est le traitement que l'on va lui appliquer, qui détermine ce qu'elle est. On pourrait imposer de renvoyer dans chaque ligne de texte TOUTES les informations propres à un fichier. Dans ce cas, resterait à déterminer l'ordre de présentation des champs : le nom, la taille, la date de création, etc. De plus en procédant ainsi on figerait le contrat qui précise la position de telle ou telle information.

Tant qu'on ajoutera de nouveaux champs aucun problème, par contre dès que l'on supprimera des champs tous nos scripts dépendants de cette présentation seront à revoir. Sans compter l'accès à une même information dans des langues différentes, ceci implique que la position des colonnes n'est pas fixe, bref un beau sac de nœuds...

On comprend qu'avec cette manière de procéder la notion de présentation des informations est tout aussi importante que leur accès. Notre ligne de texte est donc une généralité, elle contient «tout et n'importe quoi». Définie ainsi, à la hussard, une chaîne de caractères n'est pas la structure la plus appropriée pour porter des informations spécifique à un objet, bien qu'elle ait été une solution pendant de nombreuses années et le restera encore. De plus, hors contexte, c'est-à-dire sans savoir quelle commande l'a produite on ne sait pas ce qu'elle contient, si une suite de caractères et pas plus.

Premier contre-point : sous PowerShell on spécifie QUOI récupérer et pas COMMENT le récupérer.

La simple commande suivante nous renvoie une liste d'objet fichier :

```
Get-ChildItem C:\Temp\*.tmp  
#ou  
Dir C:\Temp\*.tmp  
#DIR est un alias de Get-ChildItem
```

Pour une commande fonctionnellement identique, celle-ci nous renvoie une liste d'objets fichier, et chaque objet fichier permet d'accéder à TOUTES ses informations.

We want information, information, information.

On ne reçoit plus une liste de chaînes de caractères, mais une liste d'objets et ce, sans procéder à une analyse de texte, et c'est l'utilisateur qui décide comment présenter ces informations, il n'est plus limité par les instructions du shell car chaque information lui est directement accessible. Bien évidemment PowerShell propose une présentation, un affichage, par défaut pour chaque classe d'objet recensé, il ne tient qu'à vous de la modifier si elle ne vous convient pas. Mais ceci est un autre sujet.

Nous avons dit qu'un objet peut être vu comme un conteneur regroupant à la fois des informations et des actions. Les actions nous renseignent sur les capacités qu'un objet a d'effectuer tel ou tel traitement. Pour un objet fichier on peut le copier, le supprimer ou le déplacer.

Dans un script batch (.bat, .cmd) qui est un langage de script procédural, les données et les traitements restent dispersés, d'un côté des données et de l'autre des traitements. Toujours à propos de notre objet fichier, il nous faudra déclarer autant de variables que d'informations à mémoriser, et définir autant de sous-programmes que de traitements à effectuer. Avec cette manière de faire, on fournit des données à des traitements. On utilisera des sous-programmes auxquels on passera en argument le nom du fichier sur lequel on effectuera un traitement.

Avec un objet, regroupant des données ET des traitements, on demande directement à l'objet d'effectuer telle ou telle action sans avoir à préciser les données sur lesquelles ce traitement doit se faire puisque notre objet héberge ses propres données et sait comment y accéder. Il est d'une certaine manière autonome, certes au prix d'une occupation mémoire plus importante, mais largement compensé par la facilité d'utilisation.

Second contre-point : Avec l'objet l'accès aux informations est généralisé et global. Elles sont toutes accessibles au même moment et de manière uniforme.

Notre notion de fichier ne change pas entre son usage dans un fichier batch ou dans un script PowerShell. Un nom de fichier reste un nom de fichier, un traitement de suppression aussi.

On regroupe tout ce qui a trait à la notion de fichier dans une structure de donnée dédiée indépendante du reste, on parle aussi d'encapsulation.

2.1 Comment manipuler un objet ?

Pour accéder aux données ou aux traitements, on utilise la convention d'accès suivante :

```
NomObjet.NomDeMembre
```

```
MonFichier.Taille  
MonFichier.Copier()
```

Le premier mot est le nom de l'objet (*MonFichier*), le point est un séparateur et le second mot est soit le nom d'une information (*Taille*), soit le nom d'une action (*Copier*).

En programmation objet on regroupe les termes *information* et *action* dans celui plus général de **membre**. Ceux-ci étant respectivement appelés propriété et méthode.

Un objet contient zéro ou plusieurs membres et appartient à une seule classe. Notez qu'une classe ne possédant aucun membre ne nous sera pas d'une grande utilité pratique, mais restera utile dans le domaine de la conception. Par exemple dans le Framework dotnet la classe **Object** contient bien quelques membres, mais ne définit aucun objet *particulier*, mais la notion d'objet en *général*.

Le regroupement des caractéristiques et actions dans un objet de la classe fichier concerne donc, d'après la définition précédente, les membres suivants :

```
Taille  
Copier()  
Type*
```

* Informations portant sur la classe de l'objet.

Le nom de l'objet est porté par une variable, sous PowerShell elle débute par le signe dollar (\$), exemple :

```
$MonFichier
```

On peut accéder à une propriété soit en lecture soit écriture :

```
#Lecture  
$MonFichier.Taille  
#Ecriture  
$MonFichier.Taille=255
```

Certaines propriétés peuvent être en lecture seule uniquement.

L'appel d'une méthode d'objet se distingue par l'usage de parenthèses qui délimiteront ses possibles paramètres :

```
$MonFichier.Deplacer("C:\Temp")
```

On utilise souvent un verbe pour déclarer une méthode et un nom commun pour une propriété.

Note : Attention les fonctions PowerShell n'utilisent pas cette convention d'appel.

Comme sous dotnet, le socle technique de PowerShell, tout est objet, les entiers comme les chaînes de caractères, on parlera souvent par abus de langage, d'objet ou de variable pour désigner la même chose.

PowerShell favorise l'interaction avec les données du système sous forme d'objets, bien que le producteur de ces données, c'est-à-dire le système, ne gère pas d'objet, mais des structures (ceci étant lié à des questions de performances, de l'OS et des machines, et d'historique lié aux outils de développement ayant cours lors du démarrage du développement de Windows NT), c'est la couche dotnet qui va nous les proposer sous forme d'objets.

On peut référencer les propriétés de ces objets en utilisant leurs noms plutôt que d'indiquer des colonnes et des lignes. Il en résulte un shell qui est nettement plus simple à employer parce qu'il permet à l'utilisateur de se concentrer sur ce qu'il veut réaliser au lieu de la façon dont il doit les réaliser.

C'est cette dernière approche que l'on trouve avec les batchs existants, voici un exemple (sous NT-W2K) pour récupérer la date système dans une variable :

```
Echo off
rem Insère la date système dans une variable
rem Récupère la date. Le paramètre /T renvoie une ligne contenant 2
"mots"
rem Le premier = le nom du jour. Le second = la date
rem On place donc le deuxième dans la variable
for /f "tokens=1-2 " %%A in ('Date /T') do set date=%%B

rem Reconstitue la date avec le séparateur "-"
for /f "tokens=1-3 delims=/" %%A in ('echo %DATE%') do set date=%%A-%%B-
%%C
```

devient sous PowerShell :

```
$Date=Get-Date -format "dd-MM-yyyy"
```

Note : Encore une fois les évolutions des batch sous XP et Windows Server 2003 proposent l'accès direct à cette information dans une variable prédéfinie nommée %Date%. Ici aussi on peut dire que c'est une facilité plutôt qu'une véritable évolution.

PowerShell est construit de telle manière que les données transitent d'une commande à une autre par le pipeline sous forme d'objet sans qu'il soit nécessaire de les transformer au format texte.

C'est seulement quand le texte est nécessaire qu'une conversion de l'objet en chaîne de caractère est faite. La présentation des données devient un traitement et n'est plus le résultat d'un traitement, ainsi on découple fonctionnellement les données de leurs traitements.

Le concept d'objet permet d'encapsuler des données et des traitements tout en permettant de présenter à l'utilisateur uniquement ce qu'il doit en connaître, certaines propriétés ou méthodes peuvent donc être inaccessible, car d'un usage interne. L'encapsulation sert à masquer une complexité qu'il n'est pas nécessaire de connaître pour utiliser un objet, et le protège de modifications intempestives

Ce qui nous intéresse dans un objet est de savoir quelles informations il fournit et pas comment elles sont organisées, mais aussi ce qu'il sait faire et pas comment il le fait. Ce «comment» pouvant changer sans impacter le code utilisant cet objet.

Powershell reprend cette approche à un niveau supérieur, par exemple le cmdlet **Get-Process** fournit des objets processus et c'est tout ce qui nous importe, obtenir la liste des processus sans avoir à se préoccuper de savoir comment un processus est organisé en mémoire ni comment récupérer cette liste à l'aide telle ou telle API Windows.

Voir aussi : <http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/Pipelining/>

3 Quelle classe !

Il existe de nombreuses classes représentant les éléments du système ou d'une application bureautique, trop parfois. C'est pour cette raison qu'elles sont classées sous dotnet dans des conteneurs appelés *espace de nom*.

Ces espaces de nom peuvent être ajoutés à un nom d'objet afin de le qualifier précisément et de lever une possible ambiguïté dans le cas, peu fréquent, où 2 classes possèdent le même nom.

On retrouve cette notion d'espace de nom lors de la création d'un objet, car sachez le un objet a un cycle de vie plus ou moins long selon l'usage que l'on en fait. Prenons en exemple la création d'un mail.

Il nous faut connaître :

- ✓ le nom de la classe,
- ✓ le nom de l'espace de nom dans lequel elle est définie,
- ✓ les possibles paramètres de son constructeur,
- ✓ et définir un nom de variable afin de référencer notre nouvel objet.

Pour créer un objet, on utilise ce que l'on nomme un constructeur, sa suppression de la mémoire se faisant par l'appel à un destructeur, le plus souvent cet appel est automatique.

3.1 Création d'objet mail

Sous dotnet une classe peut proposer plusieurs constructeurs d'objet, celui par défaut porte le même nom que le nom de la classe et ne nécessite pas de paramètres :

```
$Message = New-Object System.Net.Mail.MailMessage
```

On retrouve bien ici tous les éléments nécessaires à la création de notre objet Mail, où :

- `$Message` = le nom de la variable référençant notre nouvel objet,
- `System.Net.Mail` = le nom de l'espace de nom dans lequel la classe `MailMessage` est enregistrée,
- `MailMessage` = le nom de la classe, qui est aussi le nom du constructeur.

On utilise une variable pour connaître, le temps d'un traitement, l'adresse mémoire de notre objet. Un objet système peut exister indépendamment du shell, mais dans ce cas on ne pourra le manipuler qu'après son référencement dans une variable. Par exemple pour un processus on utilisera le cmdlet `Get-Process` :

```
$P=Get-Process "PowerShell"
```

Reste le terme **New-Object** qui par sa construction *Verbe-Nom* nous indique qu'il s'agit d'un cmdlet. D'après la documentation le verbe **New** "*créé une ressource vide qui n'est associée à aucun contenu*".

On utilisera le cmdlet **New-Object** chaque fois que l'on prend la décision de créer un objet d'une classe spécifique. La plupart du temps l'usage de variable déléguera à PowerShell la création de l'objet sans que l'on ait à préciser l'appel à un constructeur.

Les instructions suivantes créent trois objets un entier nommé *Integer*, une chaîne de caractères nommée *String* et un tableau nommé *Array* :

```
$Integer=10
$string="Message"
$array=@(1,2,3,4)
```

Puisque dans notre cas nous n'avons pas précisé de paramètres lors de la construction de notre objet Mail, il nous reste à renseigner ses propriétés, on peut déjà voir que certaines propriétés de notre objet **\$Message** sont renseignées avec des valeurs par défaut spécifiées lors de la conception de la classe :

```
$Message
```

Renseignons la propriété *Body*, c'est-à-dire le texte du message, mais en précisant que notre propriété, qui est un objet imbriqué, ne possède pas de valeur :

```
$Message.Body= $Null
```

La variable **\$Null** est une variable automatique créée par PowerShell, elle est utilisée notamment pour indiquer qu'une variable ou une propriété d'un objet ne référence plus d'objet. La création d'un objet implique l'usage d'un espace mémoire, d'une taille déterminée, et ce, à une adresse précise, voir la notion de heap ou tas.

La valeur de **\$Null** n'est pas égale à zéro ou à une chaîne de caractère vide, mais c'est le seul moyen pour indiquer qu'une variable ne référence pas ou plus d'objet. L'affectation de cette valeur permet aussi d'indiquer au garbage collector que notre objet peut être libéré afin de récupérer pour un autre usage la mémoire qu'il occupait.

Continuons de renseigner les propriétés de l'objet Mail :

```
$Message.From="adresseExpéditeur@Domaine"
$Message.Subject="Création d'un mail"
$Message.To.Add("adresseDestinataire@Domaine")
```

La propriété nommée *From* est une chaîne de caractères respectant le format d'une adresse email et celle nommée *To* est une collection d'adresses email, on peut donc émettre le mail vers plusieurs destinataires.

Il nous reste à joindre un fichier à notre mail, pour cela on doit créer un second objet que l'on ajoutera à la propriété *Mail.Attachments* qui est une collection de pièces jointes (collection précisée par la présence du **S** final dans le nom de la propriété). Ici le constructeur de la classe **Attachment** attend en paramètre un nom de fichier que l'on crée auparavant :

```
Dir "C:\windows\*.*)" > "C:\temp\resultat.txt"
$Attachment = New-Object System.Net.Mail.Attachment "C:\temp\resultat.txt"
$Message.Attachments.Add($Attachment)
```

Et enfin, on crée un troisième objet, cette fois ci pour envoyer le mail via un serveur SMTP :

```
$SmtpClient = New-Object System.Net.Mail.SmtpClient
$SmtpClient.Host = "NomDeServeurSmtp"
$SmtpClient.Send($Message)
```

Pour terminer on peut libérer explicitement les objets utilisés dans l'ordre inverse de leur création :

```
$Attachment=null  
$Message=null  
$SmtpClient=null
```

La lecture du chapitre intitulé "*Qui sort les poubelles ?*", complétera le code de cet envoi de mail.

On voit également dans cet exemple qu'un objet peut contenir d'autres objets. Ainsi on délègue la prise en charge des pièces jointes à l'objet liste *Attachments*.

Sachez qu'une classe peut ne pas proposer de constructeur public, c'est à dire accessible aux utilisateurs de la classe, c'est un choix de conception. Mais dans ce cas, elle déclare obligatoirement un constructeur d'accès privé pour des usages internes. Pour accéder à un objet, on doit bien le créer d'une manière ou d'une autre.

Par exemple sous PowerShell version 1 la classe *System.Management.Automation.ScriptBlock*, représentant un bloc de code anonyme, ne propose pas de constructeur. A l'origine les concepteurs voulaient se prémunir d'attaques par injection de code, mais je suppose que devant les besoins énoncés par les utilisateurs ils ont décidés de rendre ce constructeur public dans la version 2 de PowerShell.

3.2 La notion d'assembly

Il a été dit précédemment que les classes étaient regroupées dans des conteneurs appelés *espace de nom* qui est une organisation logique, le fichier qui contient réellement une ou plusieurs classes est appelé sous dotnet un assembly. C'est ce type de fichier qu'on devra charger en mémoire, comme une DLL peut l'être en Win32.

Deux classes distinctes peuvent donc être placées dans un même espace de nom, mais pas forcément dans le même fichier assembly.

Une fois chargé en mémoire, dans un domaine d'application, un assembly ne peut plus être déchargé, à moins de supprimer ce même domaine d'application. C'est-à-dire que sous PowerShell il restera actif jusqu'à la fin du processus PowerShell.exe.

Pour des raisons de performances et d'occupation mémoire, PowerShell ne charge que le strict minimum, notamment l'assembly nommé *System*. On peut donc être amené à charger explicitement un assembly afin d'accéder aux classes qu'il héberge :

```
# Chargement explicite d'assemblies  
[void][Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")  
[void][Reflection.Assembly]::LoadWithPartialName("System.Drawing")
```

Ces deux lignes d'instruction chargent et rendent accessibles les classes de composants graphiques.

4 Ah mes aïeux !

Troisième contre-point : Les classes aident à hiérarchiser les différents objets manipulés et c'est le concepteur qui détermine leur hiérarchie.

Il a été dit précédemment qu'un objet contient zéro ou plusieurs membres et appartient à une seule classe. Un objet est une instance de classe dès qu'il est créé, on utilise souvent l'expression "instancier un objet".

Une instance de classe est donc un exemplaire unique créé à l'aide d'un modèle que définit sa classe. Une classe contient la définition d'un objet.

On utilise aussi le mot *type* en lieu et place de classe, mais en théorie il ne s'agit pas de la même chose, voici les différents types disponibles sous dotnet :

- types classe,
- types interface,
- types tableau,
- types valeur,
- types énumération,
- paramètres de type,
- définitions de type générique et types génériques construits ouverts ou fermés,
- les pointeurs, mais seulement en code unsafe.

Si un objet est une instance d'une classe et d'une seule, toutes les classes héritent de la classe **Object**. Ce qui veut dire que les membres de la classe **Object** sont automatiquement disponibles pour tous ses descendants. On évite ainsi de dupliquer du code et des informations communes en les partageant avec les objets descendants.

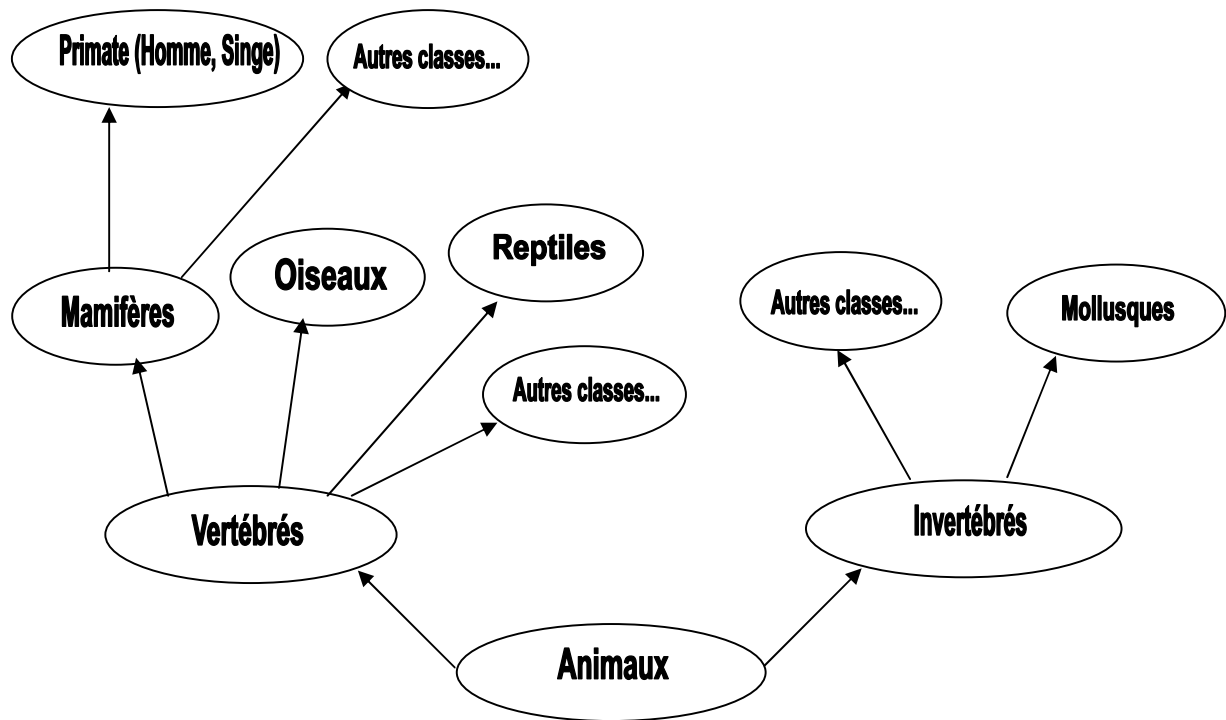
Par exemple, une bicyclette possède, entre autres, une selle, un cadre et deux roues tout comme une motocyclette. Cette dernière possède en propre un moteur, mais la définition, et la nécessaire présence, d'une selle, d'un cadre et de deux roues est identique pour ces deux objets. Je vous laisse deviner la différence qu'il y a entre ces deux objets cités et un tricycle.

Ces trois classes d'objets sont des ancêtres de la classe *cycle* : "Véhicule à deux ou trois roues mu par les jambes ou par un moteur ; ensemble des véhicules de ce type".

L'héritage est la transmission aux descendants de certains membres individuels des ascendants.

Il a été énoncé qu'une classe permettait une classification, ce qui implique un ordre de classement, tout comme le classement d'animaux en vertébrés et invertébrés. Ce type de classement est souvent représenté à l'aide d'un arbre ayant une racine et plusieurs branches, elles-mêmes constituées d'une ou plusieurs feuilles.

Représentons sommairement l'arbre d'héritage d'un être humain. Il est de la classe des primates, elle-même héritant de la classe des mammifères, qui à son tour hérite de la classe des vertébrés qui elle, finalement, définit avec la classe des invertébrés une des 2 classes primaires d'animaux :



Note : La taxinomie utilisera d'autres termes en lieu et place du terme *classe*. En anglais le mot *class* désigne, dans ce contexte, une catégorie, une sorte ou un genre.

Si on souhaite considérer sur un même plan un cheval (*mammifère*) et un corbeau (*oiseau*) on se placera sur la branche des **vertébrés** ainsi, et de ce point de vue, ils partagent des caractéristiques communes, par exemple la caractéristique : nombre de pattes.

Ou encore pour un serpent (*reptile*) et un escargot (*mollusque*), l'un possède une colonne vertébrale (*vertébré*), l'autre non (*invertébré*), dans ce cas on se placera sur la racine **Animaux** car ils ont déjà beaucoup moins de choses en commun que les deux animaux de l'exemple précédent.

Ces quatre espèces sont toutes reconnues en tant qu'espèce animale.

Inversement, si on veut voir ces animaux sur le plan de leurs spécificités, on ne peut les voir que comme un reptile ou un mollusque ou un primate ou un oiseau. Ou encore comme des vertébrés et des invertébrés.

Chacune de ces quatre espèces a suivi une évolution distincte.

Cette classification n'autorise pas d'héritage multiple, par exemple la création de chimères telle un homme oiseau ne pourrait être insérée dans cet arbre d'héritage.

On peut également adopter un autre point de vue en se basant cette fois-ci sur les comportements respectifs d'animaux. Un cheval peut courir, pour le serpent cela lui est déjà plus difficile étant apode, en revanche tous les deux ont la capacité de se déplacer. On peut donc voir dans ce contexte la course comme une spécialisation du déplacement.

Pour en revenir aux objets dans le domaine de la programmation, l'arbre d'héritage suit le même principe, mais les choix des caractéristiques servant au classement différent. Par exemple :

```
cd C:\Temp
md TestPS01
cd TestPS01

Dir C:\ >Resultat.txt
$Fichier=Dir C:\Temp\TestPS01\Resultat.txt
$Repertoire=Dir C:\Temp\TestPS0*
```

entre un fichier et un répertoire, respectivement de la classe *FileInfo* et *Directory* :

```
$Fichier.GetType().FullName
$Repertoire.GetType().FullName
```

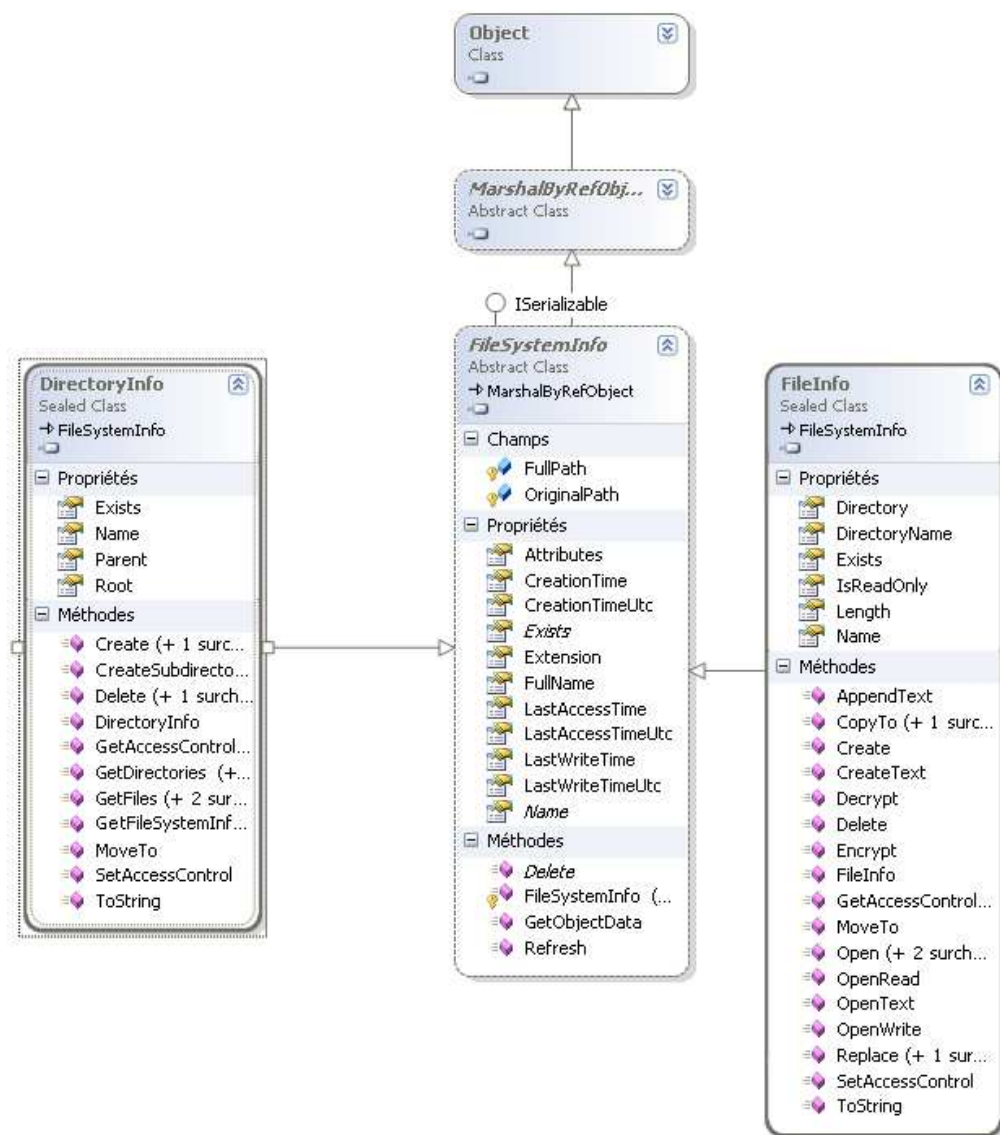
les caractéristiques communes sont principalement le nom, les attributs, les dates (création, modification, accès); d'autres ne sont pas partagées telles que les propriétés root et parent spécifique à un répertoire :

```
$repertoire,$fichier|gm|group|sort count -descending |ft
```

On peut obtenir la liste des classes héritées d'un objet comme ceci :

```
$Fichier.PSObject.tyenames ; $Repertoire.PSObject.tyenames
```

On a donc pour ces 2 classes l'arbre d'héritage suivant :



Vous trouverez plus avant dans ce tutoriel le détail des icônes accolés aux noms de membres.

Les flèches indiquent l'héritage

Partager des propriétés et des méthodes évite, lors de la conception de classes, de dupliquer du code et facilite la maintenance. L'approche objet permet donc une meilleure réutilisation.

On peut le constater dans le diagramme précédent. La classe **FileSystemInfo**, ancêtre des classes **FileInfo** et **DirectoryInfo**, porte les propriétés communes à un fichier et à un répertoire, notamment les dates de création, d'accès, ainsi que la méthode **Delete()**. Un fichier ou répertoire peut être supprimé, la manière d'effectuer l'opération sera différente selon la classe.

Notez que sur ce diagramme de classe, les classes portent d'autres informations qui ne sont pas du domaine de ce tutoriel. Enfin, sachez qu'une classe abstraite telle que **FileSystemInfo** ne peut servir à créer des instances. Ici dans un système de fichier on manipule soit un fichier soit un répertoire et rien d'autre.

La classe **FileSystemInfo** (*Abstract Class*) sert à porter les caractéristiques communes aux deux classes **FileInfo** et **Directory**. Il n'existera pas d'objet de la classe **FileSystemInfo** car elle porte uniquement une notion abstraite, toutes les classes ne donneront donc pas naissance à des objets. De plus, on ne pourra pas hériter de ces 2 classes, car elles sont dites scellées (*Sealed Class*).

4.1 Membres statiques

Une classe peut contenir des membres statiques qui ont un comportement particulier. Ces propriétés ou méthodes statiques appartiennent à la classe et pas aux objets de cette classe.

Les concepteurs peuvent autoriser l'appel à une méthode d'une classe sans imposer la création préalable d'une instance de cette classe.

Concaténons deux chaînes de caractères à l'aide la méthode statique *Join* déclarée dans la classe *String*.

L'appel suivant échoue, car la méthode statique *Join* n'est pas accessible à partir d'un objet de la classe *String* :

```
$Message= "Fichier inexistant : "  
$Message.Join("Test")
```

L'appel de la méthode a échoué parce que [System.String] ne contient pas de méthode nommée « Join ».

En revanche l'appel suivant réussit :

```
$Message = [string]::Join($Message, $Fichier.FullName)
```

car la méthode statique *Join* est accessible à partir du nom de la classe *String*.

Pour spécifier un tel appel, on utilise la syntaxe suivante :

Le nom de classe entre crochets suivi de deux caractères ':' puis du nom du membre et enfin si c'est une méthode d'une liste de paramètres.

Sachez qu'il existe des classes dites statiques, dans ce cas c'est le CLR, le runtime de dotnet, qui se charge de les créer, l'utilisateur ne peut pas les créer.

Une des limites d'une méthode statique est qu'elle ne peut accéder qu'aux données statiques de l'objet puisqu'aucune instance n'existe lors de ce type d'appel. Dans d'autres langages, on parle aussi de méthodes et de variable de classe. Sous PowerShell cet aspect ne nous concerne pas.

4.2 Surcharge de méthode

Les langages orientés objet permettent la surcharge de méthode qui n'est autre que la possibilité de définir, sous réserve de respecter certaines règles, plusieurs méthodes utilisant le même nom.

« La signature d'un membre comprend son nom et sa liste de paramètres. Chaque signature de membre doit être unique dans le type. Les membres peuvent avoir le même nom pour autant que

leurs listes de paramètres différent. Lorsque deux ou plusieurs membres d'un type représentent le même type de membre (méthode, propriété, constructeur, etc.), qu'ils possèdent le même nom, mais des listes de paramètres différentes, le membre est dit surchargé. Par exemple, la classe *Array* contient deux méthodes *CopyTo*. La première méthode prend un tableau et une valeur *Int32* tandis que la deuxième méthode prend un tableau et une valeur *Int64*. »

```
Array.CopyTo (Array, Int32)  
Array.CopyTo (Array, Int64)
```

Issu du SDK, Surcharge de membre : <http://msdn.microsoft.com/fr-fr/library/ms229029.aspx>

5 J'ai une de ces formes !

La notion de polymorphisme s'appuie sur celle d'héritage que nous venons de voir. Le polymorphisme est une caractéristique de la programmation objet qui autorise un objet à posséder plusieurs formes, c'est-à-dire à être manipulé comme une instance de sa classe ou comme une instance d'une de ses classes ancêtres, et ceci n'est valable que pour les classes de son arbre d'héritage.

On peut traiter un fichier ou un répertoire comme un objet de la classe *FileSystemInfo*, classe ancêtre commune à un fichier et à un répertoire, mais dans ce cas les propriétés et méthodes spécifiques à chacune de ces deux classes descendantes ne seront plus accessibles :

```
[System.IO.FileSystemInfo] $TypageGeneralise1=$Repertoire  
[System.IO.FileSystemInfo] $TypageGeneralise2=$Fichier  
  
[System.IO.DirectoryInfo] $TypageSpecialise1=$Repertoire  
[System.IO.FileInfo] $TypageSpecialise1=$Fichier
```

L'usage du nom de classe entre crochets déclare le type de la classe de notre variable, par défaut sous PowerShell il n'est pas nécessaire de le préciser, car il est défini automatiquement.

Essayons maintenant d'affecter l'objet **\$Repertoire** dans une variable typée en *FileInfo* :

```
[System.IO.FileInfo] $Test=$Repertoire
```

Ceci provoque l'erreur suivante :

```
"Impossible de convertir « C:\Temp\TestPS01 » en « System.IO.FileInfo »."
```

L'affectation d'un objet **\$Fichier** dans une variable typée en *DirectoryInfo* :

```
[System.IO.DirectoryInfo] $Test=$Fichier
```

```
"Impossible de convertir « C:\Tem\TestPS01\Resultat.txt » en « System.IO.DirectoryInfo »."
```

provoque la même erreur, seuls les noms de fichier et de classe sont différents dans le message d'erreur.

Dans le premier cas, lorsqu'on manipule des objets de la classe "*System.IO.FileSystemInfo*", on généralise. Ici on souhaite manipuler un objet du système de fichiers de manière générale, que cet objet soit un fichier ou un répertoire.

Dans le second cas, qui génère des erreurs, on spécialise. Là on souhaite manipuler un objet du système de fichiers de manière particulière, dans le premier exemple générant une erreur on

attend précisément un objet de la classe fichier et pas un objet d'une autre classe. Dans le second exemple on attend précisément un objet de la classe répertoire et rien d'autre.

Notre demande enfreint la règle d'héritage ce qui déclenche une erreur, car *les chiens ne font pas des chats*.

Un autre aspect du polymorphisme permet d'appeler une méthode d'un ancêtre commun à nos 2 objets du système de fichier. Ici PowerShell nous aidera un peu, car c'est un langage dynamique plus permissif qu'un langage compilé, pour qui cette démonstration avec ces 2 classes ne serait pas possible, mais le principe reste identique. Ce qu'il faut savoir est que PowerShell n'effectue aucun contrôle préliminaire sur la validité d'une propriété, le code suivant ne pose pas de problème même si la propriété *NimporteQui* n'existe pas dans la hiérarchie d'objet :

```
$Fichier.NimporteQui
```

En revanche, l'appel d'une méthode inexistante déclenchera une erreur :

```
$Fichier.NimporteQuoi()
```

L'appel de la méthode a échoué parce que [System.IO.FileInfo] ne contient pas de méthode nommée « NimporteQuoi ».

Les deux classes *FileInfo* et *Directory* proposent la méthode ***MoveTo()***, pour la première elle déplace le fichier, pour la seconde elle déplace le répertoire et tous les fichiers qu'il contient vers un nouvel emplacement.

On imagine facilement que le code de la méthode effectuant ce traitement est différent selon l'instance de classe sur laquelle elle est appelée :

```
[System.IO.FileSystemInfo] $InformationsGenerale=$Fichier
$InformationsGenerale.Moveto("C:\Temp\Resultat.txt")

[System.IO.FileSystemInfo] $InformationsGenerale=$Repertoire
$InformationsGenerale.Moveto("C:\Temp\TestPS02")
# Le répertoire courant n'existe plus
cd C:\Temp
```

Vous constaterez qu'après cette opération les propriétés des 2 objets ont été mises à jour.

Ainsi, on peut appeler une méthode de même nom, mais se comportant différemment selon la classe de l'objet.

Un autre exemple, prenons une liste d'objets graphique, des cercles, des rectangles, des triangles, chacune de ces classes a pour ancêtre la classe *Figure* qui propose la méthode *Dessine(Position)*.

L'appel, dans une boucle, de la méthode *Dessine* sur chaque élément de la collection suffira pour afficher tous les objets graphiques à l'écran :

```
#Liste d'objet de la classe Figure
ForEach ($Objet in $Liste)
{ $Objet.Dessine( $Random.GetPosition() ) }
```

Avec un langage procédural, on doit déterminer la structure de donnée qu'on souhaite manipuler puis appeler la méthode correspondante :

```
#Pseudo-code
ForEach ($Objet in $Liste)
{ if ($Objet -eq typeof(Rectangle))
    {DessineRectangle(Random.GetPosition())}
  elseif ($Objet -eq typeof(Triangle))
    {DessineTriangle(Random.GetPosition())}
  elseif ($Objet -eq typeof(Cercle))
    {DessineCercle(Random.GetPosition())}
  else Erreur("Classe inconnue.")
}
```

Dans le second cas, chaque nouvelle classe de figure géométrique implique de modifier la boucle d'affichage.

Avec l'héritage et le polymorphisme, votre code restera valide pour toutes les nouvelles classes graphiques que vous créerez. Ce mécanisme permettra de prendre en charge de nouvelles classes qui étaient inconnues au moment de la création de la hiérarchie d'origine.

Notez que PowerShell est basé objet et pas orienté objet comme on peut le lire sur certains sites. Il est basé objet, car, à la différence d'un langage de POO, il ne permet pas de créer de nouvelles classes.

5.1 Joli casting !

Il a été dit lors de l'introduction du concept d'héritage que parfois on souhaitait manipuler un objet en tant qu'objet d'une de ses classes ancêtre. Pour cela on peut utiliser une variable intermédiaire typée :

```
[System.IO.FileSystemInfo] $TypageGeneralise2=$Fichier
```

Ou utiliser un opérateur dédié nommé **-as** :

```
$TypageGeneralise2=$Fichier -as System.IO.FileSystemInfo
```

Dans ce cas le type de la variable réceptrice est automatiquement défini.

L'opérateur **-as** force la variable `$Fichier` à être considérée *comme* un objet du type précisé.

Cette opération peut échouer sans déclencher d'erreur, pour l'exemple suivant :

```
$TypageGeneralise2=$Fichier -as [System.Int32]
```

le contenu de la variable `$TypageGeneralise2` sera égal à `$Null`, car le type `System.IO.FileInfo` n'est pas un descendant du type `Int32`.

Si on souhaite être informé de l'échec de cette opération, on procédera de la manière suivante :

```
$TypageGeneralise2=[System.Int32]$Fichier
```

Cette opération est aussi appelée ***un cast***, ou transtypage en Français.

5.1.1 Comment tester si un objet est d'un type particulier ?

Il existe également l'opérateur **-is** qui permet de vérifier si un objet est d'un type particulier.

```
[System.IO.FileSystemInfo[]] $Liste=Dir C:\
ForEach ($ObjetCourant in $Liste)
{
    if ($ObjetCourant -is [System.IO.FileInfo])
        { Write-Host "Traitement fichier : $ObjetCourant"}
    else
        { Write-host "Traitement répertoire : $ObjetCourant" }
}
```

Dans cet exemple le test sur le type permet d'effectuer un traitement spécifique selon la classe de l'objet. Un pour les fichiers, un autre pour les répertoires.

Ce cas peut se présenter avec le pipeline, dans lequel transitent de nombreux objets qui peuvent être de type différent.

Dans ce dernier exemple on précise le type du tableau pour s'assurer que les objets insérés dans cette collection seront bien des descendants de la classe *FileSystemInfo*. PowerShell étant dynamique il rare de procéder ainsi, par exemple ceci suffit :

```
$Liste2=Dir C:\
ForEach ($ObjetCourant in $Liste2)
{ ..
```

PowerShell déclare implicitement un tableau d'objet, il peut donc contenir n'importe quel objet, ce qui implique que l'opération suivante est autorisée :

```
$Liste2 +=789
```

Mais elle ne l'est pas pour le premier tableau :

```
$Liste +=789
```

6 Qui sort les poubelles ?

Sous dotnet la libération mémoire se fait de manière automatique, la plupart du temps on ne s'en préoccupe pas. C'est le garbage collector du runtime dotnet qui se charge de la libération des objets inutilisés. C'est pour cette raison qu'il n'existe pas de cmdlet New-Object, il n'existe pas de cmdlet Remove-Object ou Clear-Object pour détruire un objet.

Dans certains cas on doit informer explicitement le garbage collector que l'on n'utilise plus un objet par l'appel à la méthode **Dispose()**, si toutefois la classe de l'objet propose cette méthode. Pour certaines classes l'appel à la méthode nommée **Close()** suffit, car en interne elle appellera la méthode **Dispose()**.

La libération mémoire est effectuée par le garbage collector de manière non prédictive, c'est-à-dire que l'on sait qu'il va libérer la mémoire, mais on ne sait pas quand exactement, en attendant cette libération ce comportement peut parfois bloquer une ressource système verrouillée lors de la création d'un objet ou par l'appel à une de ces méthodes.

6.1 Création d'objet mail, suite et fin.

Revenons sur notre envoi de mail (voir le chapitre "Quelle classe !"), exécutons 2 fois de suite le code complet, et ce, dans la même session PowerShell.

Lors de la seconde exécution, arrivé à la ligne de création du fichier :

```
Dir "C:\windows\*.*)" > "C:\temp\resultat.txt"
```

On obtient l'erreur suivante :

```
"Le processus ne peut pas accéder au fichier 'C:\temp\resultat.txt', car il est en cours d'utilisation par un autre processus."
```

Bien qu'ayant indiqué au garbage collector la libération des objets en affectant la valeur **\$Null** à chaque variable utilisée, le fichier reste verrouillé. En consultant la documentation de la classe *Attachement* sur MSDN (<http://msdn.microsoft.com/fr-fr/library/ms144637.aspx>), on y lit que cette classe propose une méthode **Dispose()** avec la remarque suivante :

"Appelez toujours (la méthode) Dispose avant de libérer la dernière référence à AttachmentBase. Sinon, les ressources utilisées ne seront pas libérées tant que le garbage collector n'aura pas appelé la méthode Finalize de l'objet AttachmentBase."

On doit donc explicitement libérer la référence de l'objet **\$Attachement** :

```
$Attachment.Dispose()  
# $Attachment vaut $null désormais  
$Message=$null  
$SmtpClient=$null
```

Vous pouvez exécuter de nouveau la suite d'instructions et vérifier que cette modification corrige le problème de verrou sur le fichier. Si vous placez ce code dans une fonction ou un script, les affectations de **\$Null** pour les variables sont optionnelles, car la fin de l'exécution de la fonction ou du script affectera automatiquement **\$Null** à toutes les variables déclarées localement.

Mais cela ne nous dispense pas d'appeler explicitement la méthode **Dispose()** sur tous les objets le réclamant.

Concernant l'affectation de **\$Null** à une variable référençant un objet, la libération mémoire de l'objet référencé ne se fera que s'il n'existe plus aucune référence à cet objet. Que ce référencement soit propre à un script principal, interne au Framework dotnet ou encore interne à Windows.

Dans l'exemple suivant :

```
Notepad  
$Notepad=Get-Process "Notepad"  
$Notepad=$null  
#Le process Notepad existe toujours
```

L'affectation de **\$Null** à la variable **\$Notepad** ne supprimera pas pour autant le processus référencé, car le système garde une référence interne sur ce processus.

On doit terminer le processus explicitement par un appel à la méthode **CloseMainWindow()** :

```
#On récupère de nouveau une référence sur le process
#Puisqu'après l'affectation de $null la variable ne référence plus
#le processus Notepad.exe
$Notepad=Get-Process "Notepad"
$Notepad.CloseMainWindow()
```

Ou encore à l'aide de Stop-Process

```
$Notepad=Get-Process "Notepad"
Stop-Process $Notepad.ID
```

Lorsque l'on crée un objet de la manière suivante :

```
$Message = New-Object System.Net.Mail.MailMessage
```

La seule référence qui existe sur notre nouvel objet se trouve être notre variable *\$Message*, dans ce cas l'affectation de **\$Null** indiquera au garbage collector de supprimer l'objet.

6.2 À propos de la version 2 de PowerShell

Une des évolutions de PowerShell V2 facilitera la gestion des objets «disposable», par l'utilisation du bloc Try...Catch...Finally. Le bloc Finally permet d'exécuter une suite d'instruction qu'il y ait ou non des exceptions lors de l'exécution d'un script :

```
function using
{ param($obj, [scriptblock]$sb)
  try {
    & $sb
  } finally {
    if ($obj -is [IDisposable]) {
      $obj.Dispose()
    }
  }
}
```

Un exemple d'utilisation de cette fonction :

```
using ($stream = new-object System.IO.StreamReader $PSHOME\types.ps1xml)
{
  foreach ($_ in 1..5)
  { $stream.ReadLine() }
}
```

L'instruction *\$obj -is [IDisposable]*, permet de déterminer si l'objet concerné propose le comportement de suppression explicite. IDisposable est une interface, ce mécanisme autorise l'interrogation d'un objet sur un comportement précis, on peut donc lui poser la question :

Es-tu 'disposable' ?

7 Get-Member ou comment détailler un objet.

De pouvoir manipuler des objets est appréciable bien que parfois on ne sache pas ce qu'ils sont vraiment ni ce qu'ils proposent comme membres. On peut soit consulter la documentation du SDK, soit le code source de la classe de l'objet, soit utiliser le cmdlet Get-Member. Dans un premier temps, utilisons ce cmdlet et voyons quels services il peut nous rendre.

Créons un objet String puis passons-le en paramètre au cmdlet Get-Member :

```
$s="Chaîne de caractères."
Get-member -inputobject $s

TypeName: System.String

Name      MemberType Definition
-----
Clone     Method      System.Object Clone()
CompareTo Method      System.Int32 CompareTo(Object value), System.Int32 CompareTo(String strB)...
...
Length    Property    System.Int32 Length {get;}
```

Il nous affiche la liste des membres public de l'objet `$s`. Les informations renvoyées sont :

- **TypeName** : Le nom du type de l'objet,
- **Name** : le nom du membre,
- **Membertype** : le nom de son type (méthode, propriété...),
- **Definition** : la déclaration du membre.

Pour faciliter la saisie clavier on peut utiliser différent raccourci, un sur le nom du cmdlet l'autre sur le nom d'un paramètre :

```
gm -i $s -membertype property

TypeName: System.String

Name MemberType Definition
-----
Length Property System.Int32 Length {get;}
```

Le paramètre *membertype* filtre la liste d'après une combinaison de nom de type de membre, pouvant être : *AliasProperty*, *CodeProperty*, *Property*, *NoteProperty*, *ScriptProperty*, *Properties*, *PropertySet*, *Method*, *CodeMethod*, *ScriptMethod*, *Methods*, *ParameterizedProperty*, *MemberSet*, *All* (le type *Event* est disponible à partir de PowerShell V2)

Pour le détail de type de membre, consultez le chapitre 3 du tutoriel suivant :

<http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/CreationDeMembresSynthetiquesSousPowerShell/>

Détaillons le champ *Definition* :

```
Clone     Method      System.Object Clone()
➤ System.Object est le nom de la classe de l'objet renvoyé par la méthode Clone.

CompareTo Method      System.Int32 CompareTo(Object value), System.Int32 CompareTo(String strB)...
➤ Pour cette méthode Get-Member affiche la liste des surcharges de la méthode
  CompareTo. Le code suivant :

$S.CompareTo.OverloadDefinitions
```

Affichera la liste complète des surcharges de la méthode *CompareTo* de la classe **System.String**, voir le contenu du champ *Typename* affiché par *Get-Member* pour la variable **\$s** :

```
Length      Property      System.Int32 Length {get;}
```

- **{get;}** indique une propriété en lecture seule.
- **{get; set;}** indiquera une propriété en lecture et en écriture.

Il reste possible d'afficher tous les membres de type property (*Property*, *NoteProperty*, etc.) :

```
#Récupère un tableau de fichiers
$A=dir
#Affiche le détail du dernier élément du tableau
gm -i $A[-1] -membertype *property
#Affiche le détail d'un membre étant lui-même un objet.
$A[-1].SetAccessControl|gm
```

Il est possible d'obtenir la liste des membres d'une classe en interrogeant le type de la classe :

```
[system.Enum].GetMembers()
#ou la liste de ces interfaces
[System.Net.Mail.Attachment].GetInterfaces()
```

L'appel suivant renverra toujours des informations sur la classe **System.Type**, classe permettant d'interroger d'autres classes (voir : système de réflexion de dotnet) :

```
[system.Enum]|Get-Members
#La liste des membres affichés sera identique dans ce cas
[System.String]|Get-Member
# ou pour celui-ci
#[MonAssembly.MaClasse]|Get-Member
```

Pour visualiser les membres privés qui ne sont pas listés par le cmdlet *Get-Members* on utilisera une surcharge de la méthode *GetMembers* de la classe **Type** :

```
$S.GetType().GetMembers([Reflection.BindingFlags]"Instance,NonPublic,GetField")|Select name|sort name
```

Selon le SDK (<http://msdn.microsoft.com/fr-fr/library/ms173104.aspx>) :

« **Type** est une classe de base abstraite qui permet des implémentations multiples. Le système fournira toujours la classe dérivée *RuntimeType*. »

On peut également obtenir la liste des membres statiques d'une classe :

```
$S|Get-Member -static
```

Plus d'informations sur le cmdlet *Get-Member* :

```
Get-Help Get-Member -Detail

#Obtenir de l'aide sur l'aide
help help -det
```


7.1 Un cadre dynamique

Les langages de programmation orientée objet ne permettent pas d'ajouter de nouveaux membres à un objet, car la notion de classe est une notion statique, PowerShell étant dynamique il autorise ce type de modification.

Votre prochaine mission, si vous l'acceptez, sera d'aborder ce sujet au travers du tutoriel suivant :

<http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/CreationDeMembresSynthetiquesSousPowerShell/>

8 Et j'ai crié à l'aide...

On a pu voir dans le chapitre à propos du garbage collector qu'il est parfois utile de prendre connaissance de la documentation d'une classe avant de s'en servir. De plus, Get-Member ne permet pas d'obtenir ce niveau de détail.

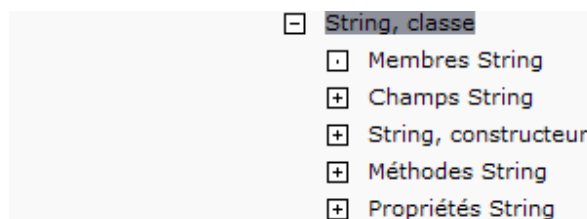
Où trouver de l'aide sur les classes du Framework dotnet ?

Ici : <http://msdn.microsoft.com/fr-fr/library/ms310244.aspx>

Voyons rapidement comment elle fonctionne :



La partie gauche de la page MSDN contient un onglet de navigation et indique dans quelle version du Framework et quel espace de nom on se trouve.



L'ouverture d'un nœud, représentant ici un nom de classe, propose soit la liste de tous ses membres, soit le détail de tous ses champs (propriétés), ou de toutes ses méthodes, etc.

La partie de droite contient la documentation d'une classe :

[MSDN](#) > [MSDN Library](#) > [Développement .NET](#) > [Versions précédentes](#) > [.NET Framework SDK 2.0](#) > [Class Library Reference](#) > [System](#) > [String, classe](#)

☐ Réduire tout Filtre de langage : C#

Bibliothèque de classes .NET Framework

String, classe

Représente du texte sous forme d'une série de caractères Unicode.

Espace de noms : System

Assembly : mscorlib (dans mscorlib.dll)

☐ Syntaxe

```
C#  
[SerializableAttribute]  
[ComVisibleAttribute(true)]  
public sealed class String : IComparable, ICloneable, IConvertible,  
    IComparable<string>, IEnumerable<string>, IEnumerable,  
    IEquatable<string>
```

Cette page est spécifique à
**Microsoft Visual Studio
2005/.NET Framework 2.0**

D'autres versions sont également
disponibles pour :

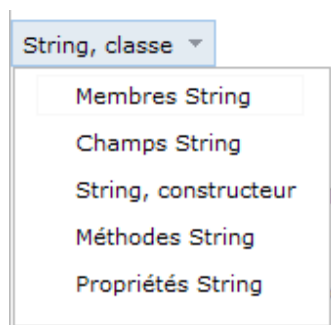
- [Microsoft Visual Studio
2008/.NET Framework 3.5](#)

Sur cet extrait de page, on retrouve en haut la position dans l'arborescence, puis le nom de classe, son espace de nom et le nom de l'assembly à charger si toutefois PowerShell ne le charge pas automatiquement. La dll *mscorlib* contient le runtime du framework dotnet.

Ensuite cette page propose la déclaration *à minima* de la classe String.

La cadre à droite précise le Framework en cours de visualisation, pour PowerShell consultez les pages spécifiques au Framework 2.0.

Si vous placez la souris sur le nom de la classe, vous pouvez accéder aux pages la détaillant :




Choisissons le menu *Membres*

Membres String

Représente du texte sous forme d'une série de caractères Unicode.


Les tableaux suivants listent les membres exposés par le type `String`.

Constructeurs publics

Nom	Description
 <code>String</code>	Surchargé. Initialise une nouvelle instance de la classe <code>String</code> .



Début

Champs publics

Nom	Description
 <code>Empty</code>	Représente la chaîne vide. Ce champ est en lecture seule.




Début

Propriétés publiques

Nom	Description
 <code>Chars</code>	Obtient le caractère à une position de caractère spécifiée dans cette instance.
 <code>Length</code>	Obtient le nombre de caractères dans cette instance.

Début

Méthodes publiques












Nom	Description
 <code>Clone</code>	Retourne une référence à cette instance de <code>String</code> .
 <code>Compare</code>	Surchargé. Compare deux objets <code>String</code> spécifiés.
 <code>CompareOrdinal</code>	Surchargé. Compare deux objets <code>String</code> en évaluant les valeurs numériques de <code>Char</code> correspondant dans chaque chaîne.

Sous PowerShell on s'intéresse uniquement aux membres publics.

Avant d'aller plus loin, examinons la signification des icônes présents sur ces pages :

Éléments graphiques

Les rubriques relatives aux références du langage incluent des icônes pour fournir les informations immédiates sur les éléments répertoriés dans les tableaux.

Graphique	Description
	Classe publique
	Interface publique
	Structure publique
	Délégué public
	Énumération publique
	Champ public
	Propriété publique
	Méthode publique
	Méthode protégée
	Prise en charge par le .NET Compact Framework
	Statique

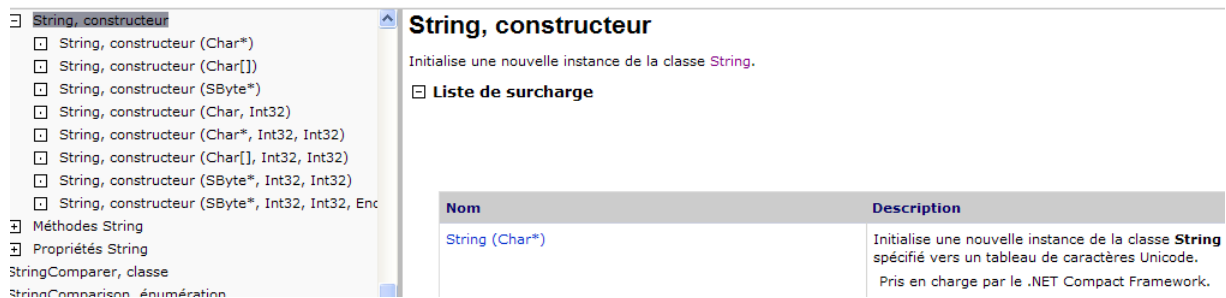
Microsoft Visual Studio
2005/.NET Framework

D'autres versions sont é
disponibles pour :

- Microsoft Visual Stud
2008/.NET Framework

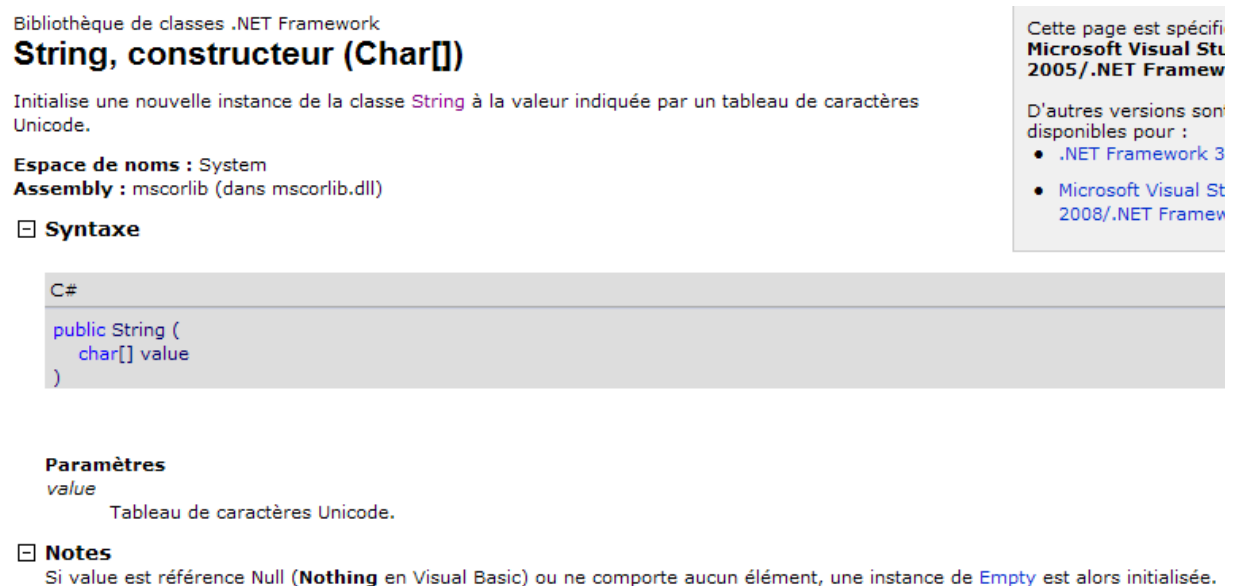
Les événements étant représentés par l'icône suivant : ⚡

Revenons sur la page des membres de la classe **String**. Cliquons sur la première entrée, section constructeur, affichée ici en bleu ([String](#)). La page affichée propose la liste des constructeurs de la classe **String** :



Nom	Description
String (Char*)	Initialise une nouvelle instance de la classe String spécifié vers un tableau de caractères Unicode. Pris en charge par le .NET Compact Framework.

Réitérons l'opération sur la seconde entrée [String\(char\[\]\)](#). Nous arrivons donc au terme de notre navigation :



String, constructeur (Char[])

Initialise une nouvelle instance de la classe **String** à la valeur indiquée par un tableau de caractères Unicode.

Espace de noms : System
Assembly : mscorlib (dans mscorlib.dll)

Syntaxe

```
C#  
public String (  
    char[] value  
)
```

Paramètres

value
Tableau de caractères Unicode.

Notes

Si *value* est référence Null (**Nothing** en Visual Basic) ou ne comporte aucun élément, une instance de [Empty](#) est alors initialisée.

On retrouve dans la partie supérieure les mêmes informations déjà présentées, ensuite on y trouve la signature de notre constructeur et le détail des paramètres ainsi que des notes relatives au comportement de ce constructeur.

La page contient également les sections suivantes :

☐ Exemple

L'exemple simple de code suivant montre comment vous pouvez créer une instance de la classe **String** avec ce constructeur.

```
C#  
  
// Unicode Mathematical operators  
char [] charArr1 = { "\u2200", "\u2202", "\u200F", "\u2205" };  
String szMathSymbols = new String(charArr1);  
  
// Unicode Letterlike Symbols  
char [] charArr2 = { "\u2111", "\u2118", "\u2122", "\u2126" };  
String szLetterLike = new String(charArr2);  
  
// Compare Strings - the result is false  
Console.WriteLine("The Strings are equal? " +  
    (String.Compare(szMathSymbols, szLetterLike) == 0 ? "true" : "false") );
```

☐ Plates-formes

Windows 98, Windows 2000 SP4, Windows CE, Windows Millennium Edition, Windows Mobile pour Pocket PC, Windows Mobile pour Smartphone, Windows Server 2003, Windows XP Édition Media Center, Windows XP Professionnel Édition x64, Windows XP SP2, Windows XP Starter Edition

Le .NET Framework ne prend pas en charge toutes les versions de chaque plate-forme. Pour obtenir la liste des versions prises en charge, consultez [Configuration requise](#).

☐ Informations de version

.NET Framework

Prise en charge dans : 2.0, 1.1, 1.0

.NET Compact Framework

Prise en charge dans : 2.0, 1.0

☐ Voir aussi

Référence

[String, classe](#)
[Mathematical Symbols](#)

L'exemple concerne ici le langage C#, une adaptation sera donc nécessaire pour effectuer la même opération sous PowerShell, sous réserve qu'elle y soit possible.

Les sections *Plates-formes* et *Informations de version* sont importantes, car elles contiennent les possibles restrictions de prise en charge selon la version de l'OS ou du Framework dotnet.

La plupart du temps il n'est pas nécessaire de consulter cette documentation mais seulement dans les cas où PowerShell ne propose pas de cmdlet facilitant la manipulation d'une classe particulière, par exemple les classes SMTP et Mail.

Selon les membres, d'autres sections peuvent exister. Notamment celle nommée **Exceptions**, prenons l'exemple de la méthode *SmtpClient.Send(String)* ([http://msdn.microsoft.com/fr-fr/library/swas0fwc\(VS.80\).aspx](http://msdn.microsoft.com/fr-fr/library/swas0fwc(VS.80).aspx))

☐ Exceptions

Type d'exception	Condition
ArgumentNullException	From est référence Null (Nothing en Visual Basic). - ou - To est référence Null (Nothing en Visual Basic). - ou - message est référence Null (Nothing en Visual Basic).
ArgumentOutOfRangeException	Il n'y a pas de destinataires dans To , CC et BCC .
InvalidOperationException	Ce SmtpClient a un appel SendAsync en cours. - ou - Host est référence Null (Nothing en Visual Basic). - ou - Host est égal à la chaîne vide (""). - ou - Port est zéro.
ObjectDisposedException	Cet objet a été supprimé.
SmtpException	La connexion au serveur SMTP a échoué. - ou -

Cette section est importante, car elle nous indique les cas d'erreur pouvant se produire lors de l'appel à cette méthode. Vos scripts et traitements devront donc les prendre en considération, enfin si vous souhaitez ne pas créer le lundi vos problèmes du vendredi...

Il existe d'autres sections qu'il n'est pas nécessaire d'aborder ici, sauf la section événement, car la version 2 de PowerShell supportera les événements :

SmtpClient.SendCompleted, événement

Remarque : cet événement est nouveau dans le .NET Framework version 2.0.

Se produit lorsqu'une opération d'envoi de courrier électronique asynchrone se termine.

Espace de noms : System.Net.Mail

Assembly : System (dans system.dll)

☐ Syntaxe

C#

```
public event SendCompletedEventHandler SendCompleted
```

☐ Notes

L'événement **SendCompleted** est déclenché chaque fois qu'un message électronique est envoyé de façon asynchrone lorsque l'opération d'envoi se termine. Pour envoyer un message électronique de façon asynchrone, utilisez les méthodes [SendAsync](#).

[SendCompletedEventHandler](#) est le délégué pour **SendCompleted**. La classe [AsyncCompletedEventArgs](#) fournit le gestionnaire d'événements avec les données d'événement.

**Microsoft Visual Studio
2005/.NET Framework 2.0**

D'autres versions sont également disponibles pour :

- [.NET Framework 3.0](#)
- [Microsoft Visual Studio 2008/.NET Framework 3.5](#)

Dans ce cas, et si besoin est, on pourra s'abonner à cet événement pour effectuer un traitement complémentaire en cas d'appel asynchrone à cette méthode.

8.1 Quelques liens utiles

Aide sur les classes WMI : [http://msdn.microsoft.com/en-us/library/aa394554\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa394554(VS.85).aspx)

Poster des classes du Framework dotnet 2.0 :

<http://blogs.msdn.com/photos/brada/picture524537.aspx>

Poster des classes du Framework dotnet 3.5 :

<http://blogs.msdn.com/pandrew/archive/2007/11/02/announcing-the-net-framework-3-5-commonly-used-types-and-namespaces-poster.aspx>

Le nombre de types selon les versions des Frameworks :

<http://blogs.msdn.com/brada/archive/2008/03/17/number-of-types-in-the-net-framework.aspx>

9 Conclusion

Paradoxalement, il n'est pas nécessaire de maîtriser tous les concepts de la programmation orientée objet avant d'utiliser PowerShell, bien que la compréhension de certains de ses mécanismes de base vous permettra d'utiliser au mieux toutes les possibilités de PowerShell, de comprendre des scripts de niveaux avancés et vous évitera de continuer à faire du batch sous PowerShell.

On peut donc voir les cmdlets comme des objets spécialisés dont la gestion est complètement intégrée dans le shell. Ces objets recevant et renvoyant des objets vous devez prendre l'habitude de faire de même au sein de vos scripts. De plus les informations externes à PowerShell vous seront le plus souvent fournies sous forme de texte structuré ou non. Dans ce cas la construction d'objet personnalisé facilitera la manipulation de ces informations, c'est ce que fait le cmdlet **Import-Csv**.

Cette introduction semblera trop poussée pour certain(e)s, mais au fil de sa rédaction il était difficile de passer sous silence certains points. Tous ne sont pas détaillés, mais les éléments de réponse contenus dans ce tutoriel vous éviteront d'être noyé d'informations incompréhensibles.

Ce qu'il faut retenir est que le concept d'objet n'est pas en soi difficile à comprendre, c'est votre manière de voir les données, que vous manipulez quotidiennement, qu'il vous faudra changer.

Et ça, ça prend un certain temps ;-)

10 Liens

Constructeurs (Guide de programmation C#)

<http://msdn.microsoft.com/fr-fr/library/ace5hbzh%28VS.80%29.aspx>

PowerShell est inspiré du C# et reprend certaines de ces constructions syntaxiques, mais n'est en rien comparable, la présence du lien précédent est à titre indicatif.

Tout ce que vous avez toujours voulu savoir sur les ramasses miettes dotnet et Java

<http://www.dotnetguru.org/articles/GC/GC.html>

Destructeurs d'objet et Finaliseurs sous .NET (Notez que la version US aborde le C#)

<http://laurent-dardenne.developpez.com/articles/Delphi/2005/DotNet/Destructeurs-et-Finaliseurs/>

Un très bon livre pour débutant

L'orienté objet

<http://conception.developpez.com/livres/?page=livresObj#L9782212120844>

La taxinomie, science du classement, pourquoi et comment classer.

<http://projetconnaissance.free.fr/classement/taxinomie.html>

Taxinomie des animaux :

<http://educ.csmv.qc.ca/mgrparent/vieanimale/taxonomie2.html>

Linné

Le botaniste suédois Carl Von Linné estime que la terminologie en usage à son époque ne permet pas de nommer toutes les espèces européennes et encore moins celles découvertes dans le Nouveau Monde. Pour remédier à cela, il développe le système de classification des organismes par règne, phylum, classe, ordre, famille, genre et espèces. Chaque espèce est définie par un nom double, en latin, qui permet aux scientifiques du monde entier de la désigner sans équivoque. Ce système est toujours utilisé de nos jours.

Wiki-BIOScope, Namur http://webapps.fundp.ac.be/umdb/wiki-bioscope/index.php/Carl_von_Linn%C3%A9

L'origine des espèces, de Charles Darwin.

<http://tinyurl.com/cohaeo>