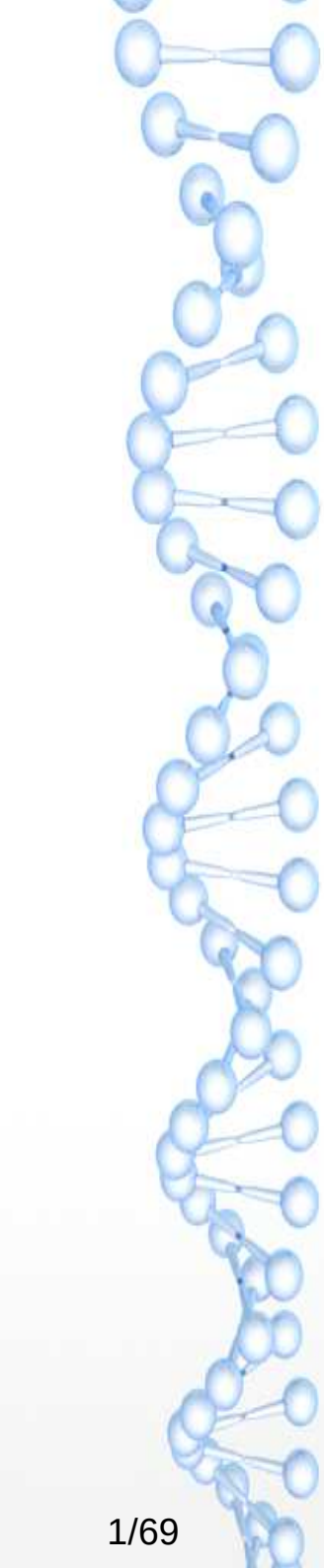
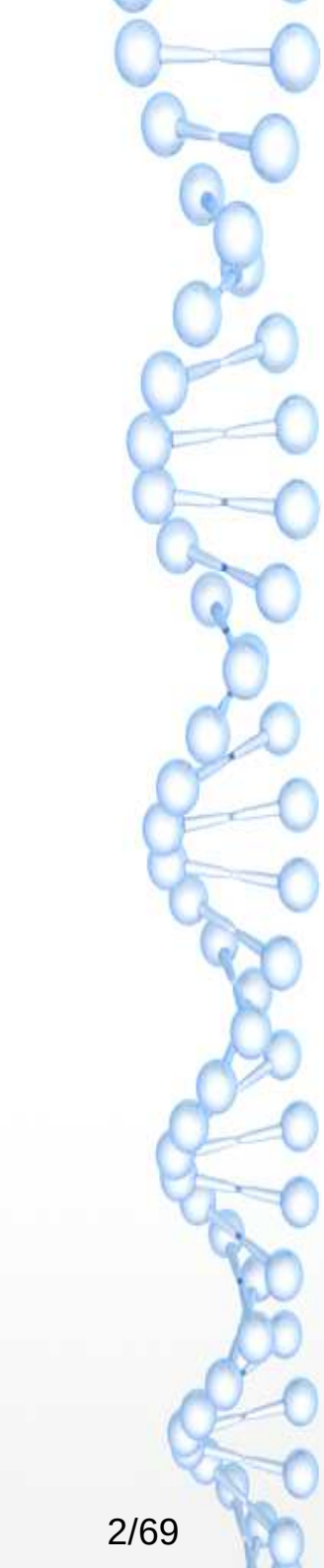


POWERSHELL

Tondeur Hervé



Windows PowerShell, anciennement Microsoft Command Shell (MSH), est une interface en ligne de commande et un langage de script développé par Microsoft.

Il est inclus dans Windows 7 et plus (y compris la version grand public) et fondé sur la programmation orientée objet (et le framework Microsoft .NET).

Depuis Windows 8, PowerShell dispose d'une place plus prononcée au sein du système d'exploitation avec un raccourci dans toutes les fenêtres de l'Explorateur de fichiers, dans le menu Fichier

Nous nous interesserons ici à la version 3.0 de PowerShell

PowerShell Historique

- Un language de script orienté objet développé par Microsoft.
- Il s'appuie sur le framework Microsoft .NET et est désormais directement intégré aux nouveaux systèmes d'exploitation Windows 7 et Windows Server 2008.
- Powershell est compatible avec toutes les versions de Windows qui supportent .NET 2.0 et +
-
- PowerShell est supporté aujourd'hui uniquement par les systèmes suivants:
  - **Windows XP Service Pack 2 & 3**
  - **Windows Server 2003 Service Pack 1**
  - **Windows Vista**
  - **Windows Seven**
  - **Windows Server 2008**
  - **Windows Server 2012**

Qu'est ce que PowerShell

- Un langage de scripting (automatique vs manuel)

  ex: creation des comptes utilisateurs.

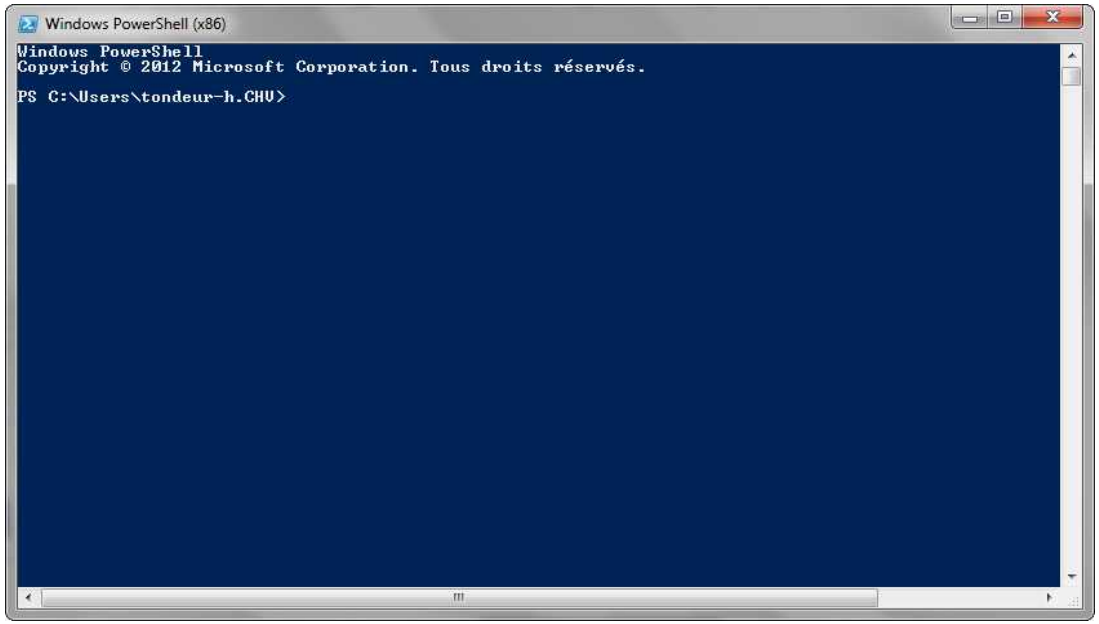
- Interpreteur de commandes.

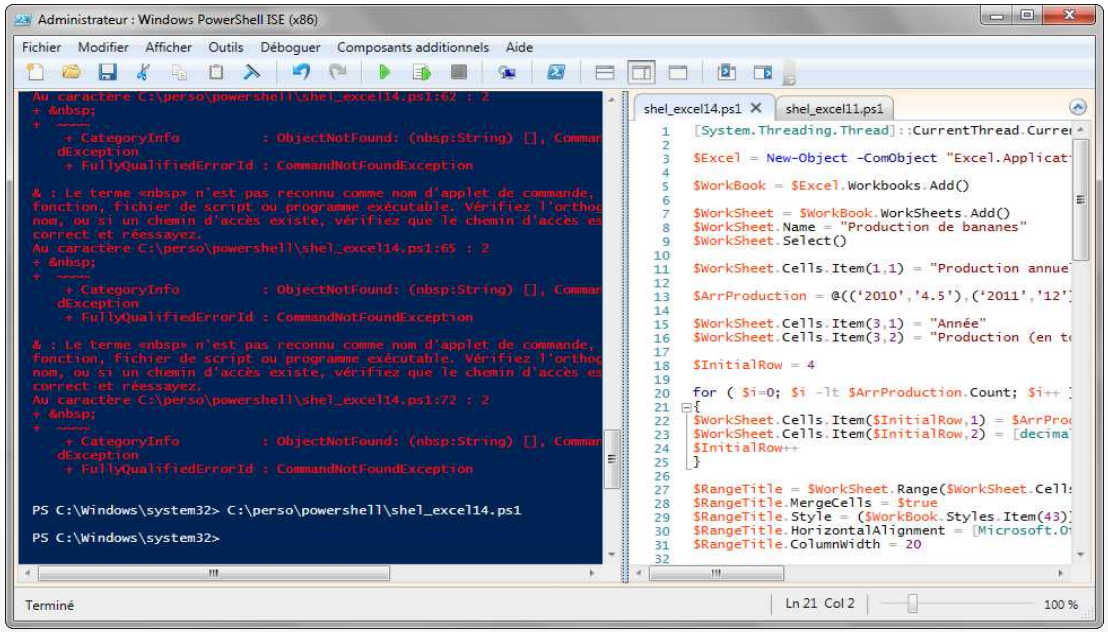
- Vous ne manipulez pas juste des textes mais le plus souvent des objets associés au Framework .Net.

- Fournis aussi avec un jeu de commandes très riches.

- Une aide en ligne integrée avec des niveaux d’explications détaillés et des exemples illustrant l’utilisation des différentes commandes.

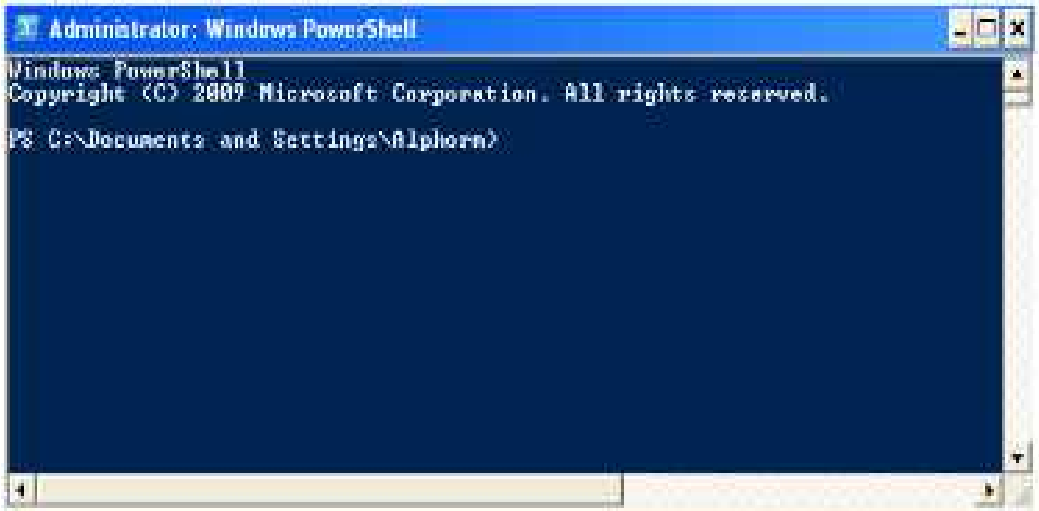
Deux manière de lancer PowerShell, soit en mode console pur ou en mode editeur (ISE).

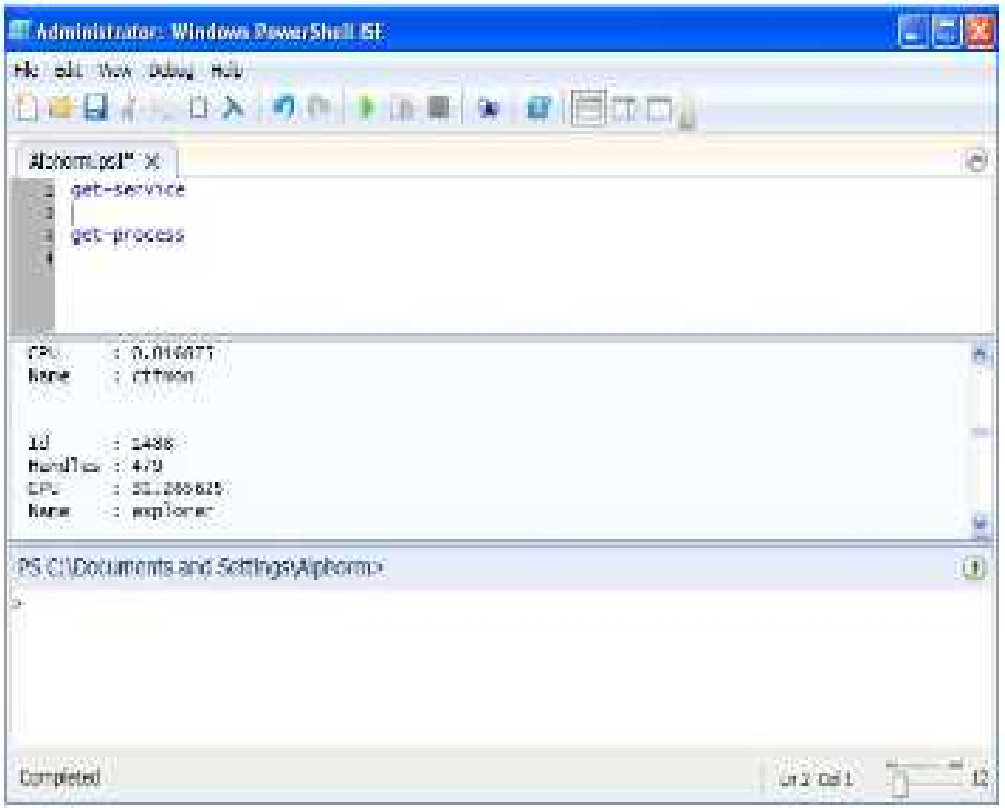




Commande	Résultat
F1	Réécris la dernière commande lettre par lettre.
F3	Auto complète une commande.
F4	Supprime jusqu'au curseur.
F5	Remonte dans l'historique des commandes.
F7	Affiche l'historique des commandes. (CTRL+F7 l'efface)
F8	Auto complète votre ligne avec votre historique.
F9	Spécifie une ligne de l'historique précise (F7)
Tab / Shift + tab	Auto complète votre commande.
Flèche haut / bas	Navigue dans l'historique des commandes. (F7)
échappe	Efface la ligne entière.
CTRL + Fleche gauche / droite	Navigue sur la ligne du prompt mot par mot.

Commande	Résultat
Alt+ espace +E	Navigue vers menu édition.
CTRL+C	Cesse l'exécution de la commande en cours.
CTRL+S	Pause l'affichage en cours.
CTRL + end	Supprime tout depuis le curseur.





Commande	Résultat
CTRL+O	Ouvre un script.
CTRL+N	Nouveau script.
CTRL+S	Sauvegarde le script.
CTRL+T	Nouvel onglet.
CTRL+W	Nouvel onglet réseau.
F5	Exécution du code en cours.
F8	Exécution d'une sélection du code.

Obtenir de l'aide avec Get-Help

- Permet de nous fournir de l’aide sur n’importe quelle commande
- Get-Help ma-command OU Help maCommande OU maCommande -?
- PowerShell permet 3 niveaux d’aides (STD, -detailled et -full).
- Help get* OU Help *-item
- Il est recommandé d’utilser Help maCommande suivi du niveau de détail (-detailed OU –full).
- L'aide est disponible également en ligne en utilisant la commande Help maCommande -online
- L’aide PowerShell est disponible aussi pour les tableaux, les opérateurs de comparaison, les boucles, les fonctions, etc.

Pipe et redirection en PowerShell

Powershell se comporte comme le shell Unix dans ce cadre là.

```
Get-Process | more
```

Le signe pipe | permet au résultat de la commande de gauche d'être transféré à la commande de droite du Pipe.

```
Get-ChildItem -Recurse c : > fichiers.txt
```

Le signe > permet de rediriger le résultat de la commande de gauche dans le fichier indiqué aprés le signe >.

```
Get-Date >> fichiers.txt
```

Il est possible d'utiliser le signe >> qui permet de réaliser une concaténation du résultat de la commande de gauche avec le contenu du fichier indiqué.

Les Command-Lets (cmdlet) ou Applets de commande

C'est le nom que porte les commandes PowerShell, elles sont constitué d'un verbe, suivi de '-' et d'un nom, d'un certains nombres de paramétres.

Get-Command -commandType cmdlet

Permet de lister l'ensemble des CmdLets PowerShell.

Get-Command -verb get

Permet de lister les cmdlets qui commencent par le verbe Get, il est possible d'utiliser les verbes suivants :
Add, Clear, Compare, Convert, Copy , Export, Format, Get, Group, Import , Measure, Move, New, Out, Read, Remove, Rename, Resolve, Restart, Resume, Select, Set, Sort, Split, Start, Stop, Suspend, Tee, Test, Trace, Update, Write.

L'aide de Windows PowerShell décrit les applets de commande, les fonctions, les scripts et les modules Windows PowerShell.
Elle explique les concepts tels que les éléments du langage Windows PowerShell.

Pour obtenir de l'aide sur une applet de commande, tapez :

Get-Help <nom-applet_commande>

Paramétres des commandes

Get-ChildItem c:\windows

Vous pouvez utiliser Get-Help pour obtenir l'ensemble de l'aide sur Get-ChildItem et trouver quels sont les paramétres supportés:

Get-Help Get-ChildItem -full

Ceci vous donnera des informations compréhensible et plusieurs exemples.

```
PS C:\Windows\system32> Get-Help Get-ChildItem -full

NOM
    Get-ChildItem

SYNTAXE
    Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [-Include <string[]>] [-Exclude <string[]>] [-Recurse] [-Force] [-Name]
    [-UseTransaction] [-Attributes <FlagsExpression[FileAttributes]> {ReadOnly | Hidden | System | Directory | Archive | Device |
    Normal | Temporary | SparseFile | ReparsePoint | Compressed | Offline | NotContentIndexed | Encrypted | IntegrityStream |
    NoScrubData}] [-Directory] [-File] [-Hidden] [-ReadOnly] [-System]  [<CommonParameters>]

    Get-ChildItem [[-Filter] <string>] -LiteralPath <string[]> [-Include <string[]>] [-Exclude <string[]>] [-Recurse] [-Force] [-Na
    [-UseTransaction] [-Attributes <FlagsExpression[FileAttributes]> {ReadOnly | Hidden | System | Directory | Archive | Device |
    Normal | Temporary | SparseFile | ReparsePoint | Compressed | Offline | NotContentIndexed | Encrypted | IntegrityStream |
    NoScrubData}] [-Directory] [-File] [-Hidden] [-ReadOnly] [-System]  [<CommonParameters>]


PARAMÈTRES
    -Attributes <FlagsExpression[FileAttributes]>

        Obligatoire ?                    false
        Position ?                       Nommé
        Accepter l'entrée de pipeline ?      false
        Nom du jeu de paramètres             (Tout)
        Alias                            Aucun(e)
        Dynamique ?                           true

    -Directory

        Obligatoire ?                    false
        Position ?                       Nommé
        Accepter l'entrée de pipeline ?      false
        Nom du jeu de paramètres             (Tout)
        Alias                            ad, d
        Dynamique ?                           true

    -Exclude <string[]>
```

Aliases: Donner un autre nom aux commandes

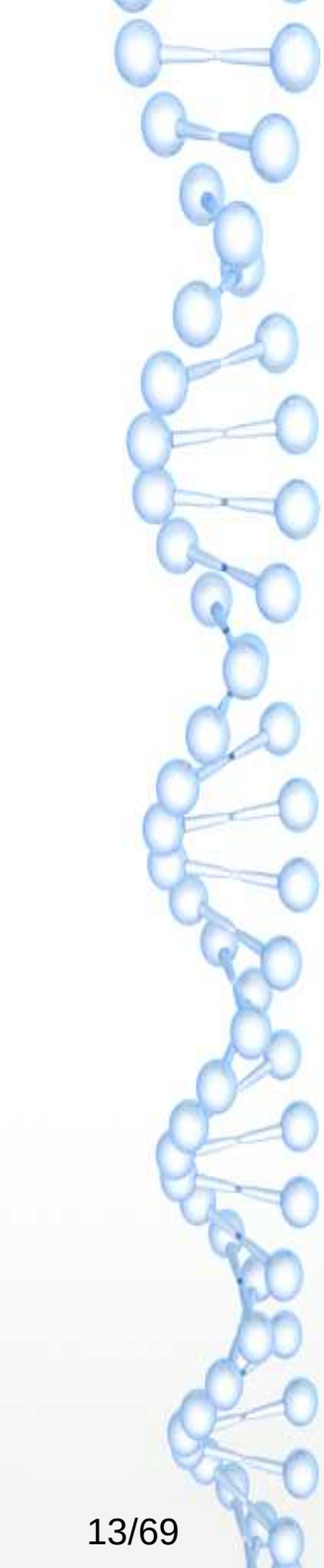
PowerShell possede des allias construit dans le système, il y a un certains nombre de ces alias de commandes déjà prette.
C'est pour cette raison que les administrateurs Windows ou Unix s'y retrouvent avec PowerShell pour lister le contenu d'un répertoire par exemple.
Il y a des alias prédéfini appelé « dir » et « ls » qui pointent tous les deux vers la cmdlet Get-ChildItem.

Get-ChildItem c:\
Dir c:\
ls c:\

Ces 3 commandes sont reconnus par PowerShell et donnent le même résultat.

Les alias ont 2 buts dans PowerShell:

• **Historique: L**es nouvelles commandes sont utilsable sous l'ancienne convention de nom.
• **Comfort:** Fles commandes fréquemment utilisées sont accessibles au travers d'une commande courte et conviviale.



Résoudre les alias

Utilisez cette commande si vous voulez connaitre la véritable commande caché derriere un alias :

$alias:Dir
Get-ChildItem

$alias:ls
Get-ChildItem

$alias:Dir liste les élements *Dir du conteneur alias:*

Dir alias :

Créer vos propre Alias

Tous le monde peut créer un alias, la commande Set-Alias permet d'ajouter un alias dans la liste des alias.

Exemple tapez les commande suivante :
edit
Set-Alias edit notepad.exe
edit

Supprimer l'alias

del alias:edit

Les Scripts PowerShell

Un script PowerShell doit avoir pour extension « .ps1 », il contient les commandes, les fonctions et les routines nécessaires à sont fonctionnement. Un script PowerShell doit être codé sous format ASCII.

Exécution d'un script :

.*test.ps1*

File "C:\Users\UserA\test.ps1" cannot be loaded because the execution of scripts is disabled on this system. Please see "get-help about_signing" for more details.
At line:1 char:10 + .\test.ps1 <<<<

Vous recevez un message d'erreur ! sous PowerShell l'éxécution de scripts est interdit, ils ne peuvent être éxécuté. Il faut donc autorisé cette éxécution en donnant votre permission avec la commande *Set-ExecutionPolicy*:

Set-ExecutionPolicy RemoteSigned

Les Délimiteurs

- Utilisation des guillemets :
- Pour delimiter les chaines de caracteres, on utilise les guillemets : simple ou double. (“ ” ou ‘ ’).
- Les guillemets double, permettent d'interpréter les variables.
- Les guillemets simple, n'interprettent pas les variables.
- Write-Host ‘Salut ..’ === Write-Host “Salut ..”
- Et les variables ?

$var1 = “Salut “ et $var2=“Hervé”
Write-host “$var1 $var2” => Salut Hervé
Write-host ‘$var1 $var2’ => $var1 $var2

Les Variables sous PowerShell

PowerShell crée les variables automatiquement, il n'est donc pas nécessaire de déclarer spécifiquement les variables, c'est PowerShell qui détermine le type de la variable lors de l'assignation, il est possible de récuperer ce type avec la commande getType().Name.

Simplement assigner une valeur a une variable.

La seule chose a connaître est que le nom de la variable est toujours préfixé par le signe "$".

```
# Creé une variable et assigner une valeur
$amount = 120
$VAT = 0.19

# Calculer:
$result = $amount * $VAT

# Remplacer des variables dans une chaine par des valeurs:
$text = "Net amount $amount matches gross amount $result"
```

Assigner et retourner une valeur

L'opérateur "=" affecte une valeur à une variable. Vous pouvez également assigner le retour d'une commande à une variable :

```
$listing = Get-ChildItem c:\
$listing
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\
Mode LastWriteTime Length Name
---- ------------ ------ ----
d---- 06.26.2007 15:36 2420
d---- 05.04.2007 21:06 ATI
d---- 08.28.2006 18:22 Documents and settings
d---- 08.08.2007 21:46 EFSTMPWP
d---- 04.28.2007 02:18 perflogs
(…)
```

```
# Temporairement stocker le résultat d'une commande externe:
$result = ipconfig
$result
```

```
Windows IP Configuration
Ethernet adapter LAN Connection:
Media state
. . . . . . . . . . . : Medium disconnected
Connection-specific DNS Suffix:
Ethernet adapter LAN Connection 2:
Media state
. . .
```

Affecter plusieurs variables en même temps

Il est possible d'affecter plusieurs variables avec la même valeur.

$a = $b = $c = 1

Affecter différentes valeurs à plusieurs Variables

$Value1, *$Value2 =* 10,20
$Value1, *$Value2 = $Value2*, *$Value1*

Lister les variables

PowerShell conserve un tableau de toutes les variables affectées, accéssible via la lettre virtuelle « variable: »

Dir variable:

Vous pourrez visualiser les variables que vous avez créé, mais beaucoup d'autres également (variables automatiques).

Opérateurs d’assignation

Operateur	Signification
=	Assigniation.
+=	Assignation an gardant la valeur precedente.
-=	Sous trait en reasignant le resultat.
/=	Divise en reasignant le resultat.
%=	Divise et assigne le modulo à la variable.

Verifier qu'une variable existe

Il est possible de verifier cela avec la commande Test-path, celle retourne une valeur booleenne.

Test-path variable:\maVariable

Supprimer une Variable

```
creer une variable test:
$test = 1
# verifier que la variable existe:
Dir variable:\te*
# supprimer la variable:
del variable:\test
# la variable est supprimé de la liste:
Dir variable:\te*
```

Quelques cmdlets pour la gestion des Variables

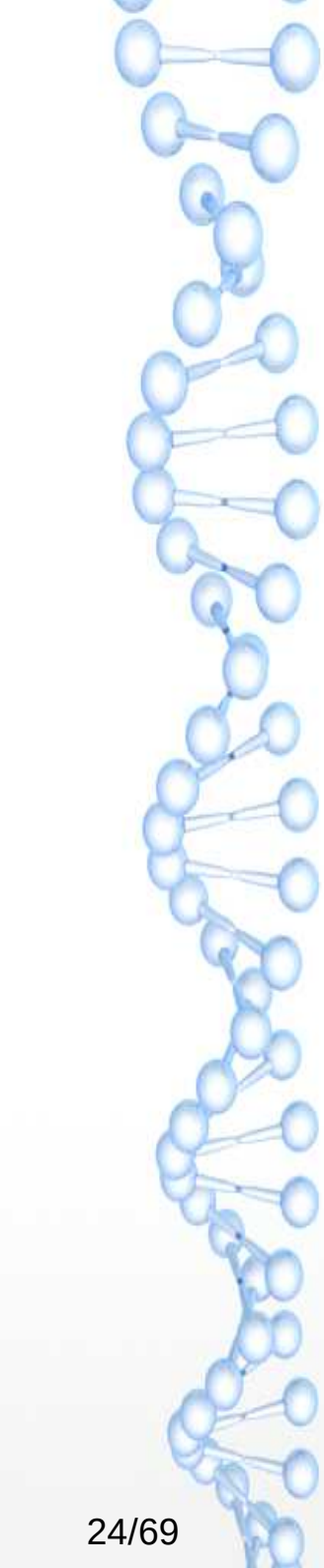
CmdLet	Description
Clear-variable	Vide le contenue d'une variable, mais ne détruit pas cette variable, elle prend la valeur NULL
Get-Variable	Récupére l'objet variable
New-Variable	Creer une nouvelle variable et permet d'affecter des informations complémentaires a celle ci
Remove-Variable	Supprime la variable
Set-Variable	Affecte une valeur ou des infos a une variable

Les opérateurs arithmétiques sur les variables

- Liste : +, -, *, / et %
- 4+5*6 est différent de (4+5)*6
- (7%3 ==> 1). Modulo
- $var1+$var2*$var3 …somme de variables.
- $ch1=‘A’ et $ch2=‘B’ … $ch1+$ch2 =➔ AB. (concaténation)

Visibilité des variables

Modificateur de visibilité	Signification
$private:test = 1	La variable est créée dans la zone de visibilité uniquement.
$local:test = 1	La variable est créée dans la zone de visibilité uniquement, elle peut être lu dans les zones de visibilités issues de la zone de visibilité de notre variable.
$script:test = 1	Visible uniquement dans le script, mais visible dans l'ensemble de ce script
$global:test = 1	Visible dans l'ensemble du script et de tous les scripts appelant



Creer des constantes

Une constante sous PowerShell, est une variable qui est protégé en écriture.

```
# Creer une variable protégée en écriture:
New-Variable test -value 100 -option ReadOnly
$test
100

$test = 200
The variable "test" cannot be overwritten since it is a constant or read-only.
At line:1 char:6 + $test <<<< = 200
```

Description sur une variable

Une variable sous PowerShell, est un objet, elle peut donc contenir des méta-Informations, comme par exemple une description.

```
# Creer une variableavec description:
New-Variable myvariable -value 100 -description "test variable" -force

#Visualiser la description :
dir variable:\myvariable | Format-Table Name, Value, Description
-autosize
```

Les Variables d'environnement

En utilisant env:, vous demandez a PowerShell de ne pas regarder la variable windir dans le conteneur des variables normale de PowerShell, mais dans les variables d'environnement de Windows.

Exemple :

$env:windir
C:\Windows

Rechercher une variable d'environnement

PowerShell conserve une copie des variables d'environnement dans une unitée virtuelle *env:* .
Donc, si vous voulez lister toutes les variables d'environnement, il faut lister le Drive env :

Dir env:

Name Value
---- -----
Path *C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:*
TEMP *C:\Users\TOBIAS~1\AppData\Local\Temp*
ProgramData *C:\ProgramData*
PATHEXT *.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;*

Dir env:userName

Créer, modifier et supprimer des variables d'environnement

Vous pouvez créer une nouvelle variable d'environnement, comme si vous crériez une variable normale.
Il faut juste spécifier le DRIVE dans lequel vous allez créer la variable, avec *env:*

```
$env:TestVar = 12
```

Vous pouvez modifier une variable d'environnement en assignant simplement une valeur à celle ci.

```
$env:OS = "Apple MacIntosh OS X"
```

Supprimer une variable d'environnement se fait de la même maniére.

```
del env:\windir
$env:windir
```

Modifier définitivement une variable d'environnement

```
$newValue = ";c:\myTools"
[environment]::SetEnvironmentvariable("Path", $newValue, "User")
```

Quand vous fermer et relancer Powershell, les changements sont conservé.
Vous pouvez verifier comme ceci:

```
$env:Path
```

Le changement n'est effective que pour vous (utilisateur logger). Si vous voulez changer cela pour tous les utilisateurs, il faut remplacer « user » par « machine » (Vous devez posséder les priviléges administrateur.

Récuperer le type d'une variable

```
$a=12
$a.getType().Name
```

```
Int32
```

Assigner un Type Fixe

Pour assigner un type particulier a une variable, il faut l'indiquer entre des crochets devant le nom de la variable.

Par exemple, si vous savez qu'une variable doit être compris entre 0 et 255, vous pouvez créer explicitement cette varaible avec le type Byte :

```
[Byte]$flag = 12
$flag.GetType().Name
```

```
Byte
```

Liste des types PowerShell

[ARRAY]	[PSOBJECT]
[BOOL]	[REGEX]
[BYTE]	[SBYTE]
[CHAR]	[SCRIPTBLOCK]
[DATETIME]	[SINGLE],[FLOAT]
[DECIMAL]	[STRING]
[DOUBLE]	[SWITCH]
[GUID]	[TIMESPAN]
[HASHTABLE]	[TYPE]
[INT16]	[UINT16]
[INT32]	[UINT32]
[INT64]	[UINT64]
[NULLABLE]	[XML]

Créer des tableaux sous PowerShell

$array = 1,2,3,4

$array = 1..4

$array = @*(1,2,3,"Hello")*
$array = *"Hello"*, *"World"*, 1, 2, (*Get-Date*)
Tableaux polymorphiques
Les parenthéses qui entourent la commande Get-Date, permettent d'éxécuter cette commande et d'en récupéré le résultat dans le tableau, sans ces parenthéses, c'est le texte Get-Date qui sera stocké dans le tableau.

$array = @(12)
$array = ,12
$array.Length =>1
Tableau avec un seul élément.

$array = @*()*
$array.Length => 0
Tableau avec zéro éléments

Adresser les éléments d'un tableau

Chaque élément d'un tableau est adressé en utilisant son numéro d'index.
Les numéros d'index négative compte a partir du premiér élément de la fin du tableau.
Vous pouvez également utilisez des expressions pour calculer la valeur de l'index :

```
$array = -5..12
# Acces au premier élément:
$array[0]
-5

# Acces au dernier élément (plusieurs méthodes):
$array[-1]
12

$array[$array.Count-1]

$array[$array.length-1]
```

Choisir plusieurs éléments dans un tableau

Vous pouvez utiliser les crochets pour selectionner plusieurs élélments d'un tableau. En faisant cela vous obtenez un nouveau tableau contenant uniquement les éléments sélectionnés :

```
# Stocke le listing des répertoires dans une variable:
$list = dir

# Affiche seulement les 2ieme, 5ieme, 8iem, et 13ieme entrées:
$list[1,4,7,12]
```

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\Users\
Tondeur Herve
Mode     LastWrite      Time      Name
----     -------------  ------    ----
d----    07/26/2007     11:03     Backup
d-r--    08/20/2007     07:52     Desktop
d-r--    08/12/2007     10:21     Favorites
d-r--    04/13/2007     01:55     Saved Games
```

Opérations sur les tableaux

Les tableaux contiennent toujours un nombre fixe d'éléments. Vous devez donc créer une copie de ce tableau avec une nouvelle taille, pour ajouter ou supprimer un élément. Vous pouvez utiliser pour cela l'opérateur "+=" :

```
# Ajouter une nouvelle valeur à un tableau existant :

$array += 34
```

Il est possible également d'additionner deux parties d'un tableau ensemble, pour permettre le rétrait d'éléments.

```
$array=0..10
$array = $array[0..2] + $array[5..10]
$array.Count
```

9

A connaître : Lors de l'éxécution des commandes powershell, quand le résultat fait plusieurs lignes, elles retournent un tableau d'éléments, ou chaque ligne est un élément du tableau.

Et pour les tableaux multidimensionnels ?

```
#Création du tableau

$infos = ('Pierre',10),('Paul',20),('Jacques',30)

#Boucle foreach pour le parcourir
Foreach($info in $infos){
$prenom = $info[0]
$age = $info[1]
"Voici $prenom qui a $age ans. Bonjour $prenom !"
}
```

Hash Tables ou tableaux associatives

On utilise @{} pour créer une liste associative.

```
$list = @{Name = "PC01"; IP="10.10.10.10"; User="Tondeur Hervé"}

# Acces a la clé "IP" retourne la valeur assignée :
$list["IP"]
10.10.10.10
$list["Name", "IP"]
PC01
10.10.10.10

# Une clé peut également être spécifié avec une notation point:
$list.IP
10.10.10.10
# Une clé peut être également stocké dans une variable:
$key = "IP"
$list.$key
10.10.10.10
# le mot clé Keys retourne toutes les clés de la hash table:
$list.keys
Name
IP
User
# Une combinaison de ceci
$list[$list.keys]
PC01
10.10.10.10
Tondeur Hervé
```

Inserrer des nouvelles clés dans une table de Hashage

Si vous voulez insérer une nouvelle paire de clé-valeur. Il faut juste spécifier la nouvelle clé et la valeur à lui assigné.

```
# Inserer 2 nouvelles paire de clé-valeur:
$list.Date = Get-Date
$list["Location"] = "Maubeuge"

# Check result:
$list
Name Value
---- -----
Name PC01
Location Maubeuge
Date 02/21/2015 13:00:18
IP 10.10.10.10
User Tondeur Hervé
```

Comme il est facile d'inserrer des nouvelles clés, vous pouvez créer une hash table vide et y inserer les clés nécessaire :

```
# Creer une hash table vide
$list = @{}
# inserer les paire de clé-valeur quand nécessaire
$list.Name = "PC01"
$list.Location = "Maubeuge"
```

Formatage de l'affichage

- Dir ou Get-ChilItem :
- Permet de lister le contenu d'un répertoire avec 4 proprietés : Mode, LastWriteItem, Length et Name.
- + commandes spécifiques pour le formatage :
  - Format-List (fl) (forme liste)
  - Format-Table (ft) (forme tabulaire)
  - Format-Wide (fw) (une seule forme large table)
  - Format-Custom (fc) (forme personnalisée)
- Obtenir une seule proprieté d'un objet :
  (Get-ChildItem c:\autoexec.bat).CreationTime

Redirection

- Les interpréteurs de commandes traitent les informations selon :
  - Une entrée (code 0 : généralement le clavier)
  - Une sortie (code 1 : généralement la console)
  - Code d'erreur (code 2 : STD)

- Powershell possede d'autres opérateurs pour rediriger ces flux d'information.
  - > : redirige le flux vers un fichier (att: d' écraser le contenu)
  - >> : redirige le flux vers a la fin du fichier. Si n'existe, va etre cree.
  - 2>&1 : redirige le message d'erreur vers une sortie STD.
  - 2> : redirige l'erreur vers un fichier. S'il existe, le contenu sera écrasé
  - 2>> : redirige l'erreur en fin du fichier. S'il n'existe pas, sera crée.

Redirection & Pipeline

- Le principe est de connecter des commandes de telle sorte que la sortie de l'une devienne l'entrée pour l'autre (sous forme d'objets).
-
- Ex: get-Command | Out-File –FilePath c:\fichier.txt
-
- Get-ChilItem c:\temp |

  ForEach-Object ' {_.Get_extention().ToLower()} |

  sort-Object | Get-Unique |

  Out-File ' -FilePath c:\temp\extensions.txt –Encoding ASCII

Redirection & Pipeline

- Filtre Where-Object :
-
- Get-Service | where-object {$_.status –eq ‘Stopped’}
-
- Get-ChilItem | where-object {$_.length –gt 1000000}

Les conditions

Operateur	Convention	Description	Example	Resultat
-eq, -ceq, -ieq	« = »	Egal	10 -eq 15	$false
-neq, -cne, -ine	« <> »	Non Egal	10 -ne 15	$true
-gt, -cgt, -igt	« > »	Supérieur à	10 -gt 15	$false
-ge, -cge, -ige	« >= »	Supérieur ou égal à	10 -ge 15	$false
-lt, -clt, -ilt	« < »	Inférieur à	10 -lt 15	$true
-le, -cle, -ile	« ≤ »	Inférieur ou égal à	10 -le 15	$false
-contains	contient		1,2,3 -contains 1	$true
-notcontains	ne contient pas		1,2,3 -notcontains 1	$false

Operateur
-and
-or
-xor
-not

Les opérateurs de traitement de texte

Operateur	Significati on	Variante sensible à la casse	Variante insensible à la casse
-like	comme	-clike	-ilike
-notlike	Pas comme	-cnotlike	-inotlike
-match	Correspond	-cmatch	-imatch
-notmatch	Correspond pas	-cnotmatch	-inotmatch

Operateur	Signification
-replace	remplace
-join	concatène
-split	Sépare

Operateur	Signification
-contains	Contient
-Notcontains	Ne contient pas

If-ElseIf-Else

- IF (<condition>)
  { bloc d'instructions 1 }
  Else
  { bloc d'instructions 2 }

```
If (condition) {
# If the condition applies,
# this code will be executed
}
```

Les conditions doivent être incluses dans des parenthéses, si la condition est valide, alors le code inclus dans les accolades sera éxécuté:

```
If ($a -gt 10) { "$a is larger than 10" }
```

```
If ($a -gt 10)
{
"$a is plus grand que 10"
}
ElseIf ($a -eq 10)
{
"$a vaut 10"
}
Else
{
"$a est inférieur à 10"
}
```

Switch

```
$value = 1

Switch ($value)
{
1 { "Number 1" }
2 { "Number 2" }
3 { "Number 3" }
}
```

Number 1

```
$value = 8
```

Utilisation du $_

```
Switch ($value)
{

{$_ -le 5} { "Number from 1 to 5" }

6 { "Number 6" }

{(($_ -gt 6) -and ($_ -le 10))} { "Number from 7 to 10" }

}
```

Number from 7 to 10

```
$action = "sAVe"

Switch ($action)

{
"save" { "I save..." }
"open" { "I open..." }
"print" { "I print..." }
Default { "Unknown command" }
}
```

I save…

```
$action = "sAVe"

Switch -case ($action)
{
"save" { "I save..." }
"open" { "I open..." }
"print" { "I print..." }
Default { "Unknown command" }
}
```

Unknown command

```
$text = "IP address: 10.10.10.10"

Switch -wildcard ($text)
{
"IP*" { "The text begins with IP: $_" }
"*.*.*.*" { "The text contains an IP " + "address string pattern: $_" }
"*dress*" { "The text contains the string " + "'dress' in arbitrary locations: $_" }
}

The text begins with IP: IP address: 10.10.10.10
The text contains an IP address string pattern:
IP address: 10.10.10.10
The text contains the string 'dress' in arbitrary
locations: IP address: 10.10.10.10
```

```
$text = "IP address: 10.10.10.10"
Switch -regex ($text)
{
"^IP" { "The text begins with IP: " + "$($matches[0])" }
"\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}" {"The text contains an IP address " +"string
pattern: $($matches[0])" }
"\b.*?dress.*?\b" { " The text " + "contains the string 'dress' in " + "arbitrary
locations: $($matches[0])" }
}
The text begins with IP: IP
The text contains an IP address string
pattern: 10.10.10.10
The text contains the string 'dress' in
arbitrary locations: IP address
```

ForEach-Object *donne acces a chaque objet d'un pipeline*.

```
Get-Process notepad | ForEach-Object { $_.Kill() }
```

```
Get-Process notepad | ForEach-Object {
$time = (New-TimeSpan $_.StartTime (Get-Date)).TotalSeconds;
if ($time -lt 180) {
"Stop process $($_.id) after $time seconds...";
$_.Kill()
}
else {
"Process $($_.id) has been running for " + "$time seconds and have not be
stopped."
}
}
```

Boucle ForEach

- Foreach ($<element> in $<collection>)
  {
  bloc d'instructions
  }

```
# Create your own array:
$array = 3,6,"Hello",12

# Read out this array element by element:
Foreach ($element in $array) {"Current element: $element"}
```

Current element: 3
Current element: 6
Current element: Hello
Current element: 12

Comment choisir entre Foreach-Object et Foreach ?
Les régles suivantes peuvent être déduites :

- **ForEach-Object: Si vous devez acquerir le résultat rapidement et que cette acquisition est longue, alors utilisez ForEach-Object, ce qui permet d'obtenir l'information rapidement.**

- **Foreach: Si le résultat est déjà disponible dans une variable ou un tableau ou si sont acquisition est rapide.**

Do … While

```
Do {
$input = Read-Host "Your homepage"
} While (!($input -like "www.*.*"))
```

```
# Open a file for reading:
$file = [system.io.file]::OpenText("C:\autoexec.bat")

# Continue loop until the end of the file has been reached:
While (!($file.EndOfStream)) {
# Read and output current line from the file:
$file.ReadLine()
}

# Close file again:
$file.close
```

Boucle For

- For (<initialisation>;<condition>;<incrémentation>)
  {
  bloc d'instructions
  }

```
# Create random number generator
$random = New-Object system.random

# Output seven random numbers from 1 to 49
For ($i=0; $i -lt 7; $i++) {
$random.next(1,49)
}
```

Sortir d'une boucle prématurément

Utilisation de l'instruction « break »

```
While ($true)
{
$password = Read-Host "Enter password"
If ($password -eq "secret") {break}
}
```

Sauter des itérations dans une boucle

Utilisation de l'instruction « continue »

```
Foreach ($entry in Dir $env:windir)
{
# si l'élément courant match le type désiré,
# continuer immédiatement avec l'élément suivant:
If (!($entry -is [System.IO.FileInfo])) { Continue }
"File {0} is {1} bytes large." -f $entry.name, $entry.length
}
```

Fonction

Déclaration de la fonction :

```
Function MaFonction
{
# Déclaration des paramètres
        param([string]$Param1, [string]$Param2, [int]$Param3)

# Traitement métier de la fonction
        write-host "Param 1 = "  $Param1  " – Param 2= "  $Param2  " – Param 3
= "  $Param3

Return $Param1
}
```

Invocation de la fonction :
Nb :Ne pas mettre de parenthèses et virgules

```
MaFonction "P1"   "P2"    3
```

Lister les fonctions
Dir function :

Supprimer une fonctrion

Del *function*:MaFonction

Passage d'arguments a une fonction

- A**rguments Arbitraires: la variable** *$args* variable contient tous les arguments passés à la fonction. C'est une bonne solution pour implémenter des arguments optionnels.

- A**rguments Nommés:** Une fonction peut assigner un nom fixe à un argument.

- A**rguments Prédéfinis:** Les arguments peuvent inclure des valeurs par défaut. Si l'appelant ne spécifi pas de vleur à ses arguments, la valeur par défaut sera prise en compte.

- A**rguments Typés:** Les arguments peuvent être spécifiés un type particulier pour être sur que l'argument correspond à un type attendu.

$args: Arguments Arbitraires

C'est la maniére la plus simple de passer des arguments à une fonction en utilisant la variable $args. Cette variable contient les arguments spécifiés lors de l'appel de la fonction.

Il n'est pas obligatoire de passer un nombre ou des types d'arguments dans un ordre spécifique.
Les arguments sont obligatoire ou optionel. De plus, *$args* peut contenir une quantité infinie d'arguments.

```
function Howdy {
If ($args -ne $null) {
"Vos arguments: $args"
"Nombre d'arguments : $($args.count)"
$args | ForEach-Object { $i++; "$i. Arguments: $_" }
} Else
{
"Vous avez passé aucun arguments!"
}
}
```

```
function Add {
$Value1, $Value2 = $args
$Value1 + $Value2
}
Add 1 6
```

Arguments avec types spécifiés

Vous pouvez spécifié au parser quel type de paramétres, votre fonction peu accepter. Dans ce cas si l'utilisateur spécifie le mauvais type de données, le parser refuse l'arguments et affiche une erreur explicite.

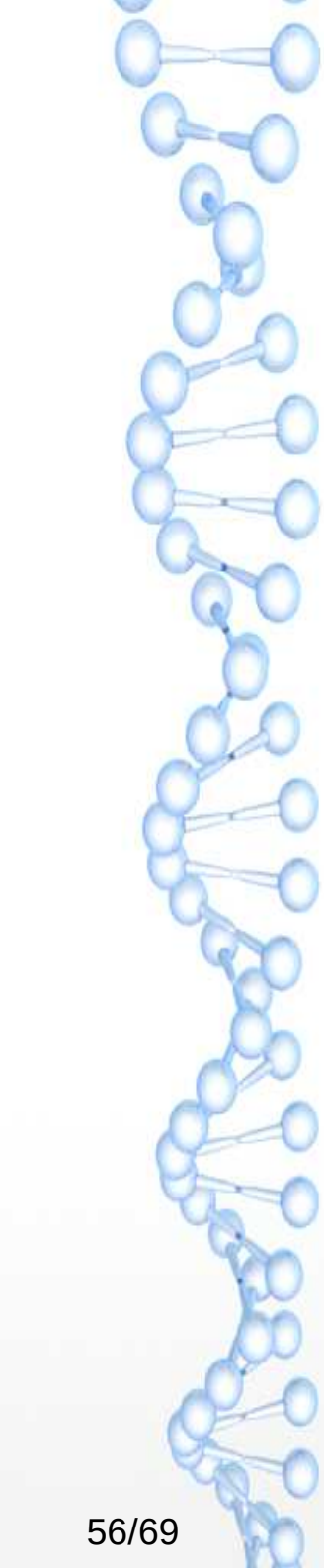
Pour cela il faut type chaque argument de la fonction dans le prototype de la fonction.

```
function Soustraire([int]$Value1, [int]$Value2)
{ $value1 - $value2 }

Soustraire 5 2
```

```
function Subtract ([double]$Value1, [double]$Value2)
{
$value1 - $value2
}
Subtract 8.2 0.2
```

Arguments avec valeurs Predefinies

```
function soustraire($Value1=10, $Value2=20)
{
    $value1 - $value2
}

function Weekday ($date=$(Get-Date))
{
    $date.DayOfWeek
}
```

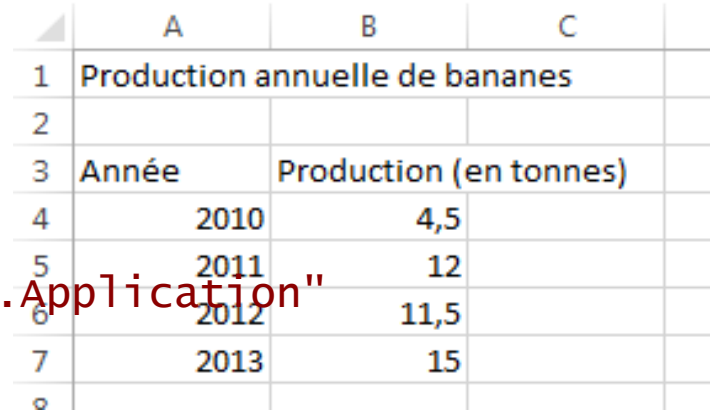


Appeler les API Excel avec PowerShell

Ouvrir un fichier Excel

```
$excel= new-object -comobject excel.application
$excel.Visible = $true
$classeur=$excel.workbooks.open("C:\users.csv")
$feuille=$excel.worksheets.item(1)

write-host $feuille.Cells.Item(1,1)

$excel.quit()
$excel=$null
```

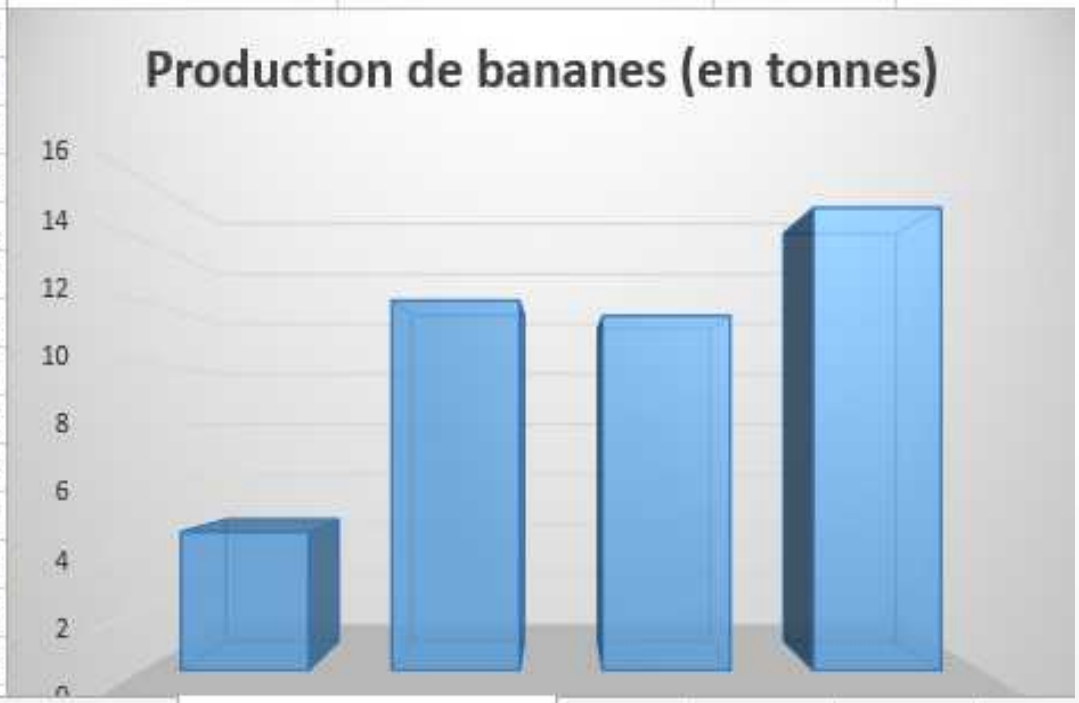
Créer un doc Excel



	A	B	C
1	Production annuelle de bananes		
2			
3	Année	Production (en tonnes)	
4	2010	4,5	
5	2011	12	
6	2012	11,5	
7	2013	15	

```
$Excel = New-Object -ComObject "Excel.Application"

$WorkBook = $Excel.Workbooks.Add()

$WorkSheet = $WorkBook.WorkSheets.Add()
$WorkSheet.Name = "Production de bananes"
$WorkSheet.Select()

$WorkSheet.Cells.Item(1,1) = "Production annuelle de bananes"

$ArrProduction = @(('2010','4.5'),('2011','12'),('2012','11.5'),
('2013','15'))

$WorkSheet.Cells.Item(3,1) = "Année"
$WorkSheet.Cells.Item(3,2) = "Production (en tonnes)"

$InitialRow = 4

for ( $i=0; $i -lt $ArrProduction.Count; $i++ )
{
$WorkSheet.Cells.Item($InitialRow,1) = $ArrProduction[$i][0]
$WorkSheet.Cells.Item($InitialRow,2) = [decimal]
$ArrProduction[$i][1]
$InitialRow++
}

$WorkBook.SaveAs("c:\temp\MaProductionDeBananes.xlsx")
$Excel.Visible = $true
```

	A	B	C	D
1	Production annuelle de bananes			
2				
3	Année	Production (en tonnes)		
4	2010	4,5		
5	2011	12		
6	2012	11,5		
7	2013	15		
8	Total	43		
9				



Production de bananes | Feuil1 | Feuil2 | Feuil3 | Feuil4

```
$RangeTitle = $WorkSheet.Range($WorkSheet.Cells.Item(1,1),$WorkSheet.Cells.Item(1,2))
$RangeTitle.MergeCells = $true
$RangeTitle.Style = ($WorkBook.Styles.Item(43)).Name
$RangeTitle.HorizontalAlignment =
[Microsoft.Office.Interop.Excel.XlHAlign]::xlHAlignCenter
$RangeTitle.ColumnWidth = 20

$RangeTable = $WorkSheet.Range($WorkSheet.Cells.Item(3,1),$WorkSheet.Cells.Item(7,2))
$ListObject = $WorkSheet.ListObjects.Add(1,$RangeTable,$null,1,$null)
$ListObject.TableStyle = "TableStyleLight6"

$ListObject.ShowTotals = $true
$ListObject.ShowHeaders = $true
$ListObject.ShowAutoFilterDropDown = $false

$RangeSort = $WorkSheet.Range($WorkSheet.Cells.Item(4,1).Address($False,$False))
$WorkSheet.Sort.SortFields.Add($RangeSort,0,1) | Out-Null
$WorkSheet.Sort.SetRange($RangeTable)
$WorkSheet.Sort.Header = 1
$WorkSheet.Sort.Apply()

$Chart = $WorkSheet.Shapes.AddChart().Chart
$Chart.ChartType = 54

try
{
$Chart.ChartStyle = 288
}
catch
{
$chart.ChartStyle = 1
}

$Chart.HasLegend = $false
$Chart.HasTitle = $true
$Chart.ChartTitle.Text = "Production de bananes (en tonnes)"

$Chart.SeriesCollection(1).XValues = $WorkSheet.Range($WorkSheet.Cells.Item(4,1),
$WorkSheet.Cells.Item(7,1))
$Chart.SeriesCollection(1).Values = $WorkSheet.Range($WorkSheet.Cells.Item(4,2),
$WorkSheet.Cells.Item(7,2))

$RangePositionChart = $WorkSheet.Range($WorkSheet.Cells.Item(10,1),
$WorkSheet.Cells.Item(25,4))
$ChartObj = $Chart.Parent
$ChartObj.Height = $RangePositionChart.Height
$ChartObj.Width = $RangePositionChart.Width
$ChartObj.Top = $RangePositionChart.Top
$ChartObj.Left = $RangePositionChart.Left
```

Lire une clé de registre

Pour lire une clé de registre, rien de plus facile :

```
PS C:\> $a = get-itemproperty -path
registry::"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion"

PS C:\> $a.ProductName
```

Ecrire une nouvelle entrée de registre

Supposons que nous voulions qu'un programme s'exécute au démarrage de la session de l'utilisateur courant. Pour information, le nom de la clé n'a aucune importance, si ce n'est d'identifier le programme.

```
PS C:\> New-ItemProperty -path HKCU:\Software\Microsoft\Windows\CurrentVersion\Run -name
Test -propertyType String -value "C:\WINDOWS\SYSTEM32\monprogramme.exe"

PSPath       :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVers
ion\Run
PSParentPath :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVers
ion
PSChildName  : Run
PSDrive         : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
Test              : C:\Windows\System32\monprogramme.exe
```

Remarque : Veuillez noter la différence de la valeur -path par rapport aux exemples précédents qui listent des clés. Nous aurions très bien pu écrire : registry::"HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run"

Valeur PropertyType	Signification
Binary	Données binaires
DWord	Nombre UInt32 valide
ExpandString	Chaîne pouvant contenir des variables d'environnement développées dynamiquement
MultiString	Chaîne multiligne
String	Toute valeur de chaîne
QWord	8 octets de données binaires

Suppression d'une clé de registre

Dans cet exemple, nous supprimons la clé nommée MonProg :

```
PS C:\> Remove-ItemProperty -Path
HKCU:\Software\Microsoft\Windows\CurrentVersion\Run -Name MonProg
```

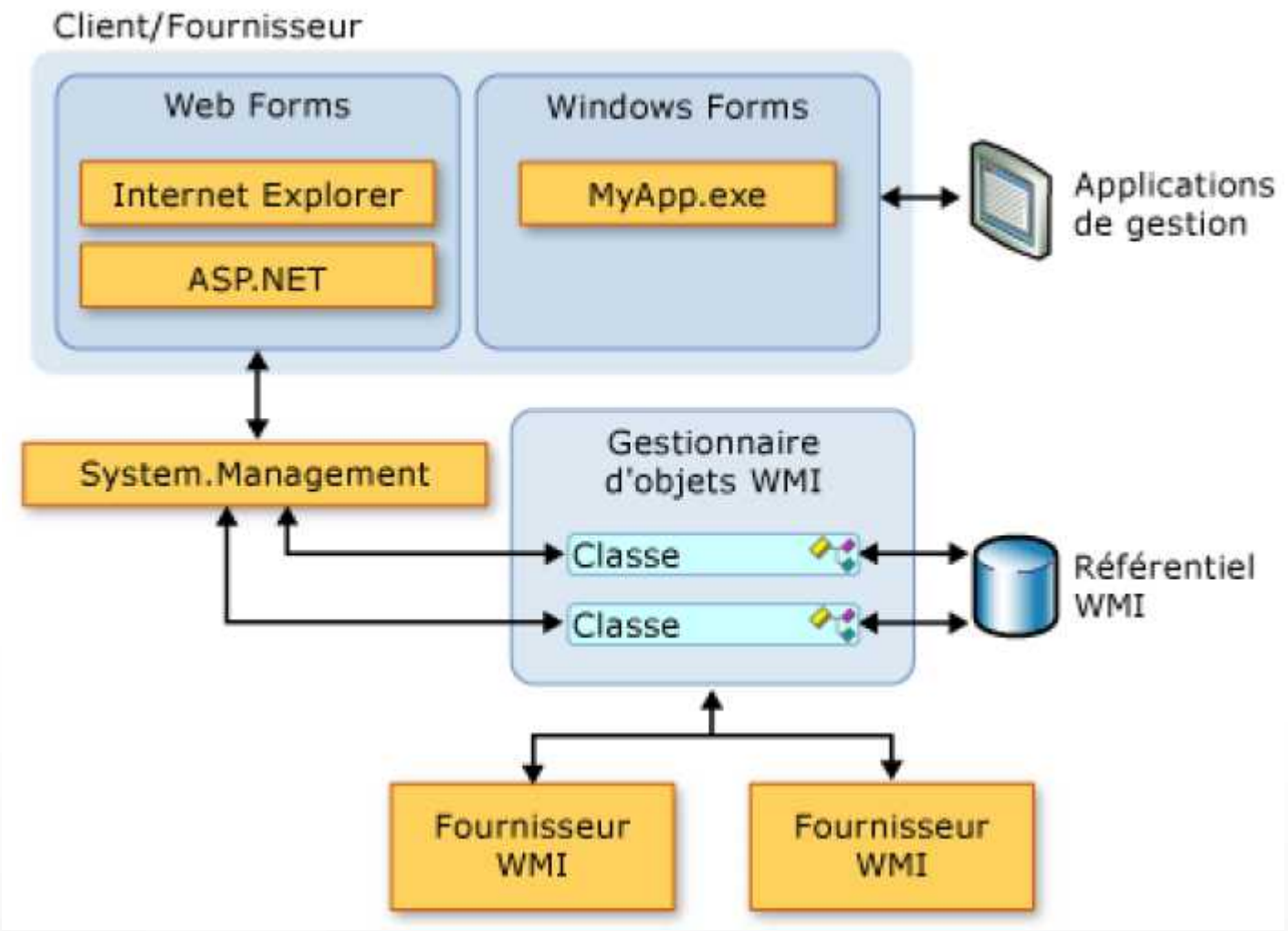
Renommer une clé de registre

Dans l'exemple précédent nous avons créé une clé de registre que nous avons appelé Test. Nous allons maintenant l'appeler MonProg :

```
PS C:\> Rename-ItemProperty -path
HKCU:\Software\Microsoft\Windows\CurrentVersion\Run -name Test
-newname MonProg
```

Si vous souhaitez avoir un retour sur le bon déroulement de la commande, vous pouvez ajouter le paramètre -PassThru.

WMI


Client/Fournisseur
Web Forms
Internet Explorer
ASP.NET
Windows Forms
MyApp.exe
Applications de gestion
System.Management
Gestionnaire d'objets WMI
Classe
Classe
Référentiel WMI
Fournisseur WMI
Fournisseur WMI

Avant tout vous ne devez pas confondre les classes WMI à proprement parlé avec celles de dotnet les encapsulant. De plus tous les objets sont à leur tour adaptés dans PowerShell afin de faciliter leur manipulation. Cette adaptation constitue la couche supplémentaire indiquée précédemment.

Prenons comme exemple la récupération d'informations sur la carte mère d'un ordinateur. Pour cela on passe en argument au cmdlet **Get-WmiObject** le nom de la classe WMI:

```
$ObjetWMI = Get-WmiObject "Win32_MotherboardDevice"
```

Avec cette syntaxe on récupère une ou plusieurs instances de classe WMI. Il n'existe qu'une seule instance de carte mère par ordinateur. Affichons le type de la classe de l'objet récupéré :

```
$ObjetWMI.GetType()
```

```
IsPublic IsSerial Name                BaseType
-------- -------- ----                --------
True     True     ManagementObject    System.Management.ManagementBaseObject
```

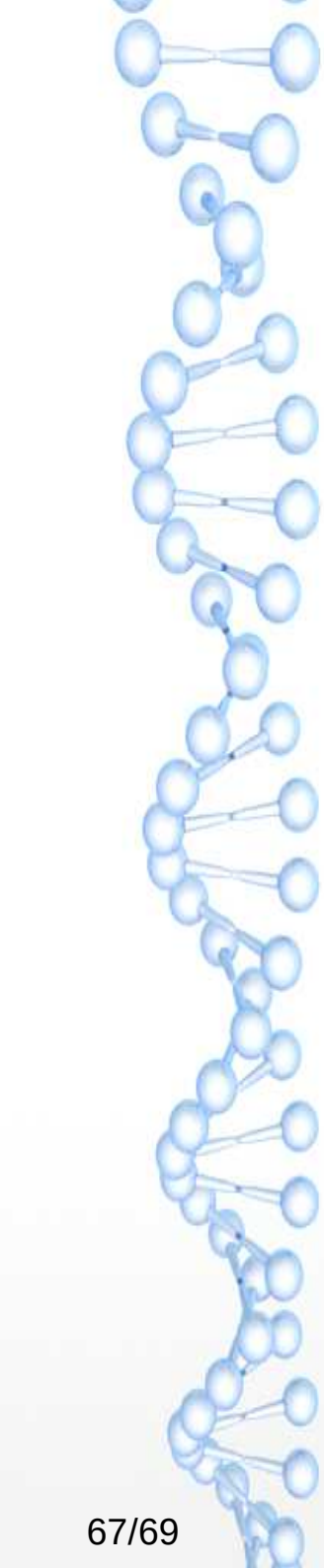
Comme vous pouvez le voir, le nom de la classe de l'objet renvoyée par Get-WMIObject est *ManagementObject* encapsulant une instance d'une classe WMI.

Vous noterez qu’avec PowerShell il n‘y pas de phase de connexion explicite au référentiel WMI cible, PowerShell s’en charge.

De plus comme il gère les collections à notre place, il n’est plus nécessaire de préciser qu’on récupère une instance ou une liste d’instances.

Certaines propriétés ou collections ne sont accessibles qu’en passant par l’objet dotnet encapsulant l’objet WMI ($ObjetWMI.psbase) et pas par l’objet adapté de PowerShell ($ObjetWMI).

Sachez aussi que Get-WmiObject renvoie des copies des instances WMI.



Où trouver des informations sur les classes WMI disponibles ?

WMI Reference : http://msdn.microsoft.com/en-us/library/aa394572(VS.85).aspx

Attention sur certaines versions de serveur, toutes les classes et provider WMI peuvent ne pas être installés par défaut et nécessiter une installation supplémentaire.

[WMI]	Accède à une instance de classe WMI particulière en précisant un chemin unique.
	`$ObjetWMI= [Wmi]'Win32_MotherboardDevice.DeviceID="Motherboard"'`
	Similaire à un appel à **Get-WMIObject** ou au code suivant :
	`$ObjetWMI= New-Object System.Management.ManagementBaseObject `
	`Win32_MotherboardDevice`

[WMICLASS]	Accède à une classe WMI notamment à ces méthodes statiques.
	`$ClassWMI=[wmiclass]"\\localhost\root\cimv2:Win32_MotherboardDevice"`
	Similaire au code suivant :
	`$ClassWMI= New-Object System.Management.ManagementClass `
	`Win32_MotherboardDevice`

[WMISEARCHER]	Exécute une interrogation WMI à l'aide du langage WQL et renvoie le résultat.
	`([wmiSearcher]'SELECT * FROM Win32_MotherboardDevice where DeviceID="Motherboard"').Get()`
	Similaire au code suivant :
	`$WMISearcher = New-Object System.Management.ManagementObjectSearcher`
	`$WMISearcher.Query ='SELECT * FROM Win32_MotherboardDevice where DeviceID="Motherboard"'`
	`$WMISearcher.Get()`

Accéder aux détails d'une classe WMI

La définition de la classe WMI est accessible par l'appel suivant :

```
$ObjetWMI=[wmiclass]"\\localhost\root\cimv2:Win32_MotherboardDevice"
$objetWMI.psbase.GetText([System.Management.TextFormat]::MOF)
```

```
[dynamic: ToInstance, provider("CIMWin32"): ToInstance,
Locale(1033):ToInstance, UUID("{8502C4BA-5FBB-11D2-AAC1-
006008C78BC7}"): ToInstance]

class Win32_MotherboardDevice : CIM_LogicalDevice
{
[read: ToSubClass, key, Override("DeviceId"): ToSubClass,
MappingStrings{"WMI"}: ToSubClass] string
DeviceID = NULL;
…
```

Un fichier MOF, **M**anaged **O**bject **F**ormat, est le fichier source contenant la description de la classe WMI.