SISR4 – Administration des Systèmes

02 le langage PowerShell

Table des matières

1 Objectifs

1.1 Objectifs du document

Ce document présente les fondements du langage PowerShell. A l'issue de cette leçon, vous devez connaître ce qu'est un langage de script, les particularités de PowerShell, les outils permettant de créer des scripts et les bases du langage PowerShell.

1.2 Qu'est ce qu'un langage de Script ?

Un langage de script est un langage composé des commandes. Ces commandes sont interprétées et exécutées directement. Elle ne font pas l'objet d'une compilation. On peut enchainer différentes commandes les une à la suite des autres et ainsi constituer un fichier que l'on peut faire exécuter par l'interpréteur de commandes. Quelques exemples :

- Le langage de commande MS-DOS.
- Le langage SQL, permettant de passer des commandes à un serveur de bases de données
- Le langage Bash utilisé dans Linux pour gérer le système.

1.3 Qu'est ce que PowerShell ?

PowerShell est un langage de script créé par Microsoft. Jusqu'à l'apparition de PowerShell, il n'existait pas de langage, dans les systèmes Windows permettant de réaliser des tâches sur le système d'exploitation autrement que par le biais d'un interface graphique à l'exception du « vieux » langage de commande MS-DOS aux capacités très limitées et de VBScript (VBS) avec l'outil d'administration WSH. Ces systèmes (VBS en particulier) présentent de très graves défaillances de sécurité.

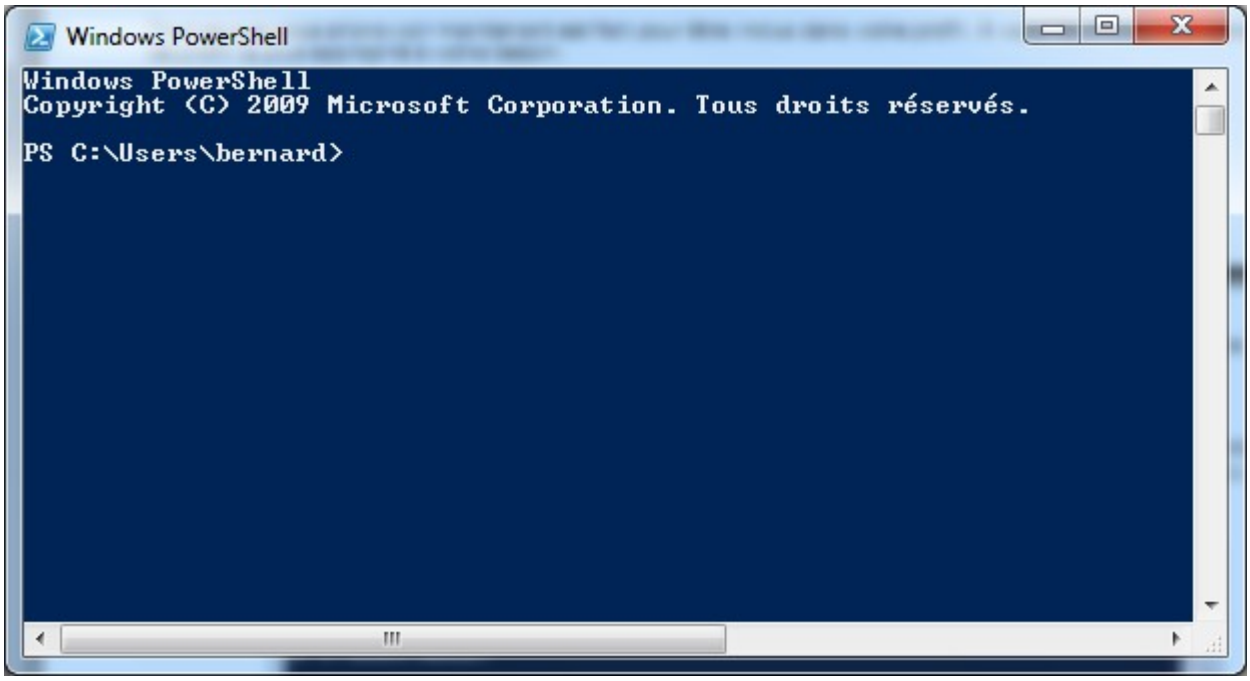
PowerShell cumule les avantages d'un interpréteur de commandes et d'un langage de script. Il est basé sur le « framework » .NET. Ce dernier est un ensemble de classes préprogrammées permettant de créer des objets grâce auxquels il est possible d'agir sur les différentes parties du système Windows. .NET se présente comme un concurrent de Java, plus spécialement dédié aux systèmes Windows. .NET existe actuellement en version 3.5 et 4.0.

PowerShell est donc un langage permettant de manipuler ces objets .NET. Il est actuellement en version 2.0 nativement avec Windows 2008 R2 et Windows 7 et en version 3.0 pour Windows 8 et Windows Server 2012. Il faut la télécharger pour les systèmes plus anciens. L'objectif de Microsoft est de placer PowerShell au coeur du système. Ainsi, par exemple, dans Exchange 2010 les actions graphiques exécutent en réalité des commandes PowerShell.

2 Les outils PowerShell

2.1 L'interpréteur de commandes

Pour utiliser PowerShell, il faut lancer l'interpréteur de commandes. Vous pouvez le trouver dans le menu démarrer, dans les accessoires. Sélectionner Windows PowerShelll et lancez le en **mode Administrateur**. Vous obtenez alors la fenêtre suivante :



Les commandes de PWS sont appelées des cmdlets (pour command-applets). Elles sont pour la plupart d'entre elles constituées de la manière suivante : un verbe, un tiret, un nom : **verbe-nom**. Par exemple

```
get-command
```

Le verbe indique l'action que l'on va appliquer sur le nom. Il y a toute une série de verbes génériques : Get,Add, Rmove, Set, etc... Les noms constituant les commandes sont toujours au singulier. C'est aussi vrai pour les options des commandes (on parle de paramètres en PWS et pas d'options).

2.2 La commande get-command

Cette commande permet de retrouver la liste de toutes les commandes existant dans PowerShell (il y en a plus de 250 !)

La liste de toutes les commandes de type cmdlet	`Get-command -commandtype cmdlet`
La liste de toutes les commandes de type cmdlet qui ont le verbe write	`Get-command -commandtype cmdlet -verb write`
La liste de toutes les commandes de type cmdlet qui portent sur le nom Object.	`Get-command -commandtype cmdlet -noun object`
Tous les alias des commandes.	`Get-command -commandtype alias`

2.3 Obtenir de l'aide sur les commandes

On peut obtenir de l'aide en permanence sur un cmdlet. Il suffit de frapper :

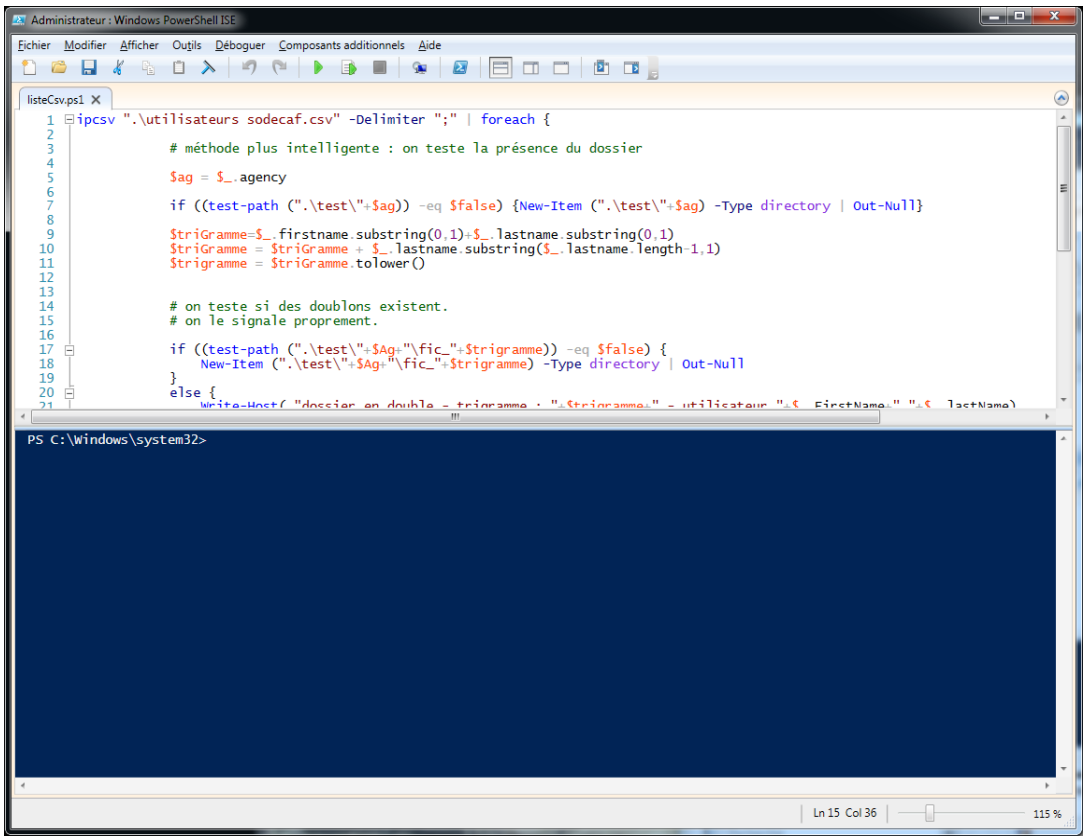
```
            get-help get-command -detailed | more

ou encore   help get-command
```

2.4 PowerShell ISE

Depuis la version 2, PowerShell est livré avec un environnement intégré d'écriture de scripts : PowerShell ISE. Celui-ci est encore très limité dans ses fonctionnalités mais il permet de :

- gérer l'édition de fichiers de scripts avec une coloration syntaxique
- enregistrer et relire plusieurs fichiers de scripts grâce à un système d'onglets.
- Lancer l'exécution et déboguer les scripts.



PowerShell ISE est composé de 2 volets dont la position peut être aménagée selon des schémas prédéfinis.

Dans la vue ci-dessus, nous pouvons remarquer :

- En bas, le volet de l'interpréteur de commandes. Les commandes frappées ici sont immédiatement exécutées. Lorsqu'une commande ou un script effectue un affichage, c'est dans ce volet qu'on trouve celui-ci ;
- En haut, l'éditeur de scripts. On remarque plusieurs onglets correspondants à chacun des fichiers de scripts en cours d'édition.

3 La notion de script

3.1 Qu'est ce qu'un script ?

Un script est un fichier contenant un ensemble de commandes. Lorsqu'on demande l'exécution du script, avec l'interpréteur approprié, chacune des commandes est exécutée l'une à la suite de l'autre, séquentiellement, tout comme dans un programme écrit avec un langage de programmation classique.

PowerShell, comme certains langages de scripts, bénéficie de fonctionnalités équivalentes à celles que l'on trouve dans les langages de programmation évolués :

- La possibilité de choisir d'exécuter ou pas une partie du script en fonction de conditions. C'est la notion d''exécution conditionnelle de type if...then ...else.
- La possibilité de répéter un ensemble de commandes un nombre fini de fois : c'est la notion de boucle que l'on trouve dans la plupart des langages.
- La possibilité de « factoriser » un certain nombre de commandes dans des fonctions. Il est ainsi plus simple d'appeler cette fonction que de répéter toujours les mêmes commandes. Les fonctions sont l'équivalent des méthodes qu'on a pu rencontrer dans les langages comme Java.

Les scripts PowerShell sont reconnaissables car ils possèdent une extension .PS1. Si vous double-cliquez sur un tel fichier, il ne s'exécute pas : il est affiché dans le bloc notes.

3.2 Exécuter des scripts PowerShell

Stratégies d'exécution (Execution Policy)

PowerShell est doté d'un système de sécurité permettant de prévenir l'exécution de scripts sans l'autorisation de l'utilisateur. Il existe 4 stratégies d'exécution, de la plus restrictive à la plus ouverte : Restricted, AllSigned ,RemoteSigned, UnRestricted, bypass et undefined

Restricted	C'est la stratégie la plus restrictive. On ne peut que frapper des commandes en mode ligne. L'exécution de scripts est interdite.
AllSigned	C'est la stratégie la plus sûre. Elle permet de n'exécuter que des scripts signés. Pour pouvoir les exécuter, il faut être en possession des certificats correspondants.
RemoteSigned	Cette stratégie permet de n'exécuter que les scripts créés localement. Les scripts téléchargés ne peuvent être exécutés si ils ne sont pas signés et si l'on n'est pas en possession des certificats correspondants.
UnRestricted	C'est la stratégie d'exécution plus ouverte. Tous les scripts peuvent être exécutés quelle que soit leur provenance. Un message d'avertissement apparaît tout de même lorsqu'on essaie de lancer un tel script.
Bypass	Rien n'est bloqué, comme dans Unrestricted, et de plus, aucun message d'avertissement n'est affiché.
Undefined	Pas de stratégie définie dans l'étendue courante.

Par défaut, toutes les stratégies d'exécution sont sur Restricted. Ce qui veut dire, qu'il est impossible

d'exécuter le moindre script powerShell sur un ordinateur Windows. Il faut débloquer les stratégies d'exécution.

Les étendues (scope)

Il est possible de restreindre la portée de la stratégie de sécurité. Il existe 3 étendues possibles de la plus petite à la plus large : Process, CurrentUser et LocalMachine.

Process	La stratégie d'exécution sélectionnée n'est appliquée que pour la session de PowerShell actuelle. Lorsque l'interpréteur de commandes est fermé, la stratégie d'exécution est perdue.
CurrentUser	La stratégie d'exécution choisie n'est valable que pour l'utilisateur actuel. Si l'on change d'utilisateur sur la même machine, elle ne s'applique plus. Il faut en définir une autre.
LocalMachine	La stratégie d'exécution choisie est valable pour tous les utilisateurs de l'ordinateur.

Par défaut, si on sélectionne une stratégie d'exécution sans définir d'étendue, l'étendue est LocalMachine.

La stratégie s'applique prioritairement est celle portant sur l'étendue process, puis celle portant sur CurrentUser et enfin celle portant sur LocalMachine.

```
          Scope                          ExecutionPolicy
          -----                          ---------------
  MachinePolicy                                Undefined
     UserPolicy                                Undefined
        Process                                Undefined
    CurrentUser                                Undefined
    LocalMachine                            RemoteSigned
```

Ici, comme les stratégies d'exécution ne sont pas définies au niveau process et au niveau currentUser, la stratégie appliquée sera celle de LocalMachine, c'est à dire RemoteSigned.

Lister les stratégies d'exécution

Pour connaître la stratégie d'exécution en cours :

```
Get-ExecutionPolicy
```

Exemple :

```
PS C:\Windows\system32> get-executionPolicy
RemoteSigned
```

Pour connaître la programmation des stratégie d'exécution sur les différentes étendues

```
Get-ExecutionPolicy -list
```

On obtient alors la liste ci-dessus.

Modifier les polices d'exécution

Il faut utiliser la commande : Set-ExecutionPolicy en utilisant les options -ExecutionPolicy et -scope.

```
PS > Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process

Modification de la stratégie d'exécution
La stratégie d'exécution permet de vous prémunir contre les scripts que
vous jugez non fiables. En modifiant la stratégie d'exécution, vous vous
exposez aux risques de sécurité décrits dans la rubrique d'aide
about_Execution_Policies. Voulez-vous modifier la stratégie d'exécution ?
[O] Oui  [N] Non  [S] Suspendre  [?] Aide (la valeur par défaut est
« O ») :
```

Il est systématiquement demandé une confirmation pour modifier la stratégie d'exécution.

Lancer l'exécution d'un script

A partir de la fenêtre de commandes MS-DOS	`PowerShell.exe -noprofile -file:c:\scripts\essai.ps1`
Dans l'interpréteur PowerShell	`C:\scripts\essai.ps1`
Dans l'interpréteur PowerShell, sur le répertoire courant	`Cd c:\scripts` `./essai.ps1`

Exécuter un script non signé provenant d'un autre ordinateur

Même si on a interdit l'exécution de script non signé provenant d'autres ordinateurs (remoteSigned ou restricted), on peut quand même débloquer cette interdiction.

Il faut cliquer avec le bouton droit, dans l'explorateur Windows, sur le fichier .ps1 correspondant à ce script, choisir « **propriétés** » dans le menu et cliquer sur le bouton « **débloquer** » sur la fenêtre des propriétés du fichier.

3.3 Inclure des scripts dans d'autres scripts

Pour inclure des scripts dans d'autres scripts, il faut utiliser le « **dotsourcing** ». Cette technique permet tout simplement d'utiliser les fonctions et les variables du script inclus comme si elles faisaient partie du script en cours. Elle consiste à faire précéder l'appel du script d'un . Suivi d'un espace

```
# Script qui contient les constantes utilisées dans le logiciel
# le . situé devant l'intruction est le "dotsourcing".
# C'est l'équivalent d'un include.
. F:\Developpements\scripts\pws\mgUser\mguConstantes.ps1
# Script qui contient les fonctions d'accès à la base de données
# Exécute la connexion à la base.
. F:\Developpements\scripts\pws\mgUser\mguSql.ps1
```

Dans l'exemple ci-dessus, les scripts mguConstantes.ps1 et mguSql.ps1 sont inclus dans le script principal. Les variables et les fonctions qu'ils contiennent peuvent être utilisés dans le script principal.

4 Les Pipeline et les filtres

4.1 Pipeline

Avec PowerShell, il est possible d'enchaîner les commandes entre elles, de telle sorte que la sortie de l'une devienne la sortie de l'autre. C'est ce qu'on appelle le pipeline. Cette possibilité est aussi donnée par d'autres langages de scripts, comme le Bash ou le langage de commande MS-DOS de Windows.

Dans PowerShell, le canal de communication ainsi créé permet de transporter des données sous forme d'objets. Le pipeline est matérialisé par le caractère « | » (AltGr 6). Il transfère le résultat d'une commande comme entrée de la commande qui lui succède. Par exemple :

```
get-command | out-file -file path c:\temp\fichier.txt
```

Dans l'instruction ci dessus, la liste des commandes disponibles issue de la commande get-command est envoyée vers la commande out-file qui la stocke dans le fichier spécifié (c:\temp\fichier.txt).

```
get-command | out-null
```

Cette commande out-null supprime immédiatement toute entrée qu'elle reçoit. Ainsi, get-command qui, par défaut envoie les données vers la console, ne renvoie plus rien du tout !

4.2 Filtre

Grâce aux pipelines, il est aisé de filtrer le résultat de certaines commandes. Un filtre se pose grâce à la commande « where-object », en abrégé : »where ».

Par exemple :

```
Get-WmiObject Win32_OperatingSystem | Get-Member -MemberType property | where {$_.name -notlike "__*"}
```

Cette commande :

- recherche un objet dans la WMI : `get-WmiObject win32_operatingSystem`
- Recherche les propriétés de cet objet : `get-member -memberType property`
- n'affiche que les propriétés ne commençant pas par les caractères « __ » : `where ($_.name...)`

5 Les éléments de base du langage

5.1 Les commentaires

Pour marquer le début d'un commentaire, PowerShell utilise le caractère #.

```
PS C:\Windows\system32> # Ceci est un commentaire
PS C:\Windows\system32> $var=12   # déclaration de la variable var
```

Dans la première ligne de cet exemple, l'ensemble de la ligne est un commentaire. Dans la 2° ligne, le commentaire ne commence qu'après le caractère #

On peut aussi réaliser des blocs de commentaires, ce qui évite de répéter le caractère # sur chaque ligne

```
PS C:\Windows\system32> <#
PS C:\Windows\system32>    Ceci est un bloc de commentaires
PS C:\Windows\system32> #>
```

5.2 Les variables

Nommage

Le nom des variables commence par le **caractère $.** Il peut contenir des chiffres et des lettres ainsi que le caractère underscore (_). Bien sur, il ne peut pas contenir d'espace. Il n'y a pas de différenciation majuscule / minuscule : $var et $VAR désignent la même variable.

Typage

Les variables n'ont pas à être explicitement déclarées et typées. C'est à leur première utilisation que PowerShell détermine leur type.

```
PS C:\Windows\system32> $var=12
PS C:\Windows\system32> $var.getType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType
```

Toutefois, on peut les définir et les typer explicitement afin de garantir leur valeur :

```
PS C:\Windows\system32> [int] $nombre = read-host "entrez un entier "
entrez un entier : Cent
Impossible de convertir la valeur « Cent » en type « System.Int32 ». Erreur :
« Le format de la chaîne d'entrée est incorrect. »
```

Dans l'exemple ci-dessus, le fait d'avoir typer explicitement la variable $nombre oblige à ce que l'on ne saisisse que des nombres.

Les types les plus utilisés :

Int	Nombre Entier
Double	Nombre réel
Char	Caractères ASCII

String	Chaine de caractères

Les variables prédéfinies

Il existe plus de 50 variables prédéfinies dans PowerShell. En voici quelques unes :

$$	Contient le dernier mot de la dernière commande frappée dans la console.
$?	Contient *true* si la dernière commande a réussi, *false* sinon.
$^	Contient le premier mot de la dernière commande frappée dans la console.
$_	Contient l'objet courant transmis par le pipe (|).
$Args	Contient le tableau des arguments passés à un script.
$Error	Contient, sous forme de tableau, la liste des erreurs de la session. $Error[0] contient la dernière erreur.
$False	Contient la valeur *False*
$Home	Contient le chemin d'accès au répertoire de l'utilisateur
$Host	Contient des informations sur l'ordinateur
$Null	Variable vide
$Pwd	Répertoire en cours
$True	Contient la valeur *true*

Vous pouvez trouver la liste complète des variables prédéfinies en frappant la commande get-variable.

Autres commandes associées aux variables

Set-variable	Définit la valeur d'une variable. Crée la variable s'il n'existe aucune variable portant le nom demandé.
New-variable	Crée une variable.
Clear-variable	Supprime la valeur d'une variable.
remove-variable	Supprime une variable et sa valeur.

Portée des variables

La portée d'une variable est la portion de code dans laquelle elle est peut être lue et modifiée. Par défaut, une variable ne peuvent être lue et modifiée que dans le bloc d'instructions où elle a été déclarée implicitement ou explicitement. C'est ce que l'on appelle la portée **LOCALE**. Une variable peut être lue dans une portée enfant de la portée initiale de la variable, mais elle ne peut pas y être modifiée :

par exemple :

```
PS C:\Windows\system32> $var = 10
PS C:\Windows\system32> function incremente { $var++ }
PS C:\Windows\system32> incremente
PS C:\Windows\system32> $var
10
```

Ici, une variable créée dans le bloc d'instruction principal d'un script ne pourra être modifiée dans une fonction, pourtant présente dans ce même script. La fonction utilise sa propre variable $var, indépendante de celle créée dans le script principal.

On peut modifier la portée d'une variable, lors de sa déclaration. Les autres portées sont **GLOBALE** et **SCRIPT**.

Dans la portée globale, la variable est accessible dans l'ensemble de la session PowerShell, jusqu'à sa destruction.

Par exemple :

```
PS C:\Windows\system32> $var=25
PS C:\Windows\system32> function incremente {$global:var++}
PS C:\Windows\system32> incremente
PS C:\Windows\system32> $var
26
```

Ici, c'est la variable globale qui a été incrémentée, non plus la variable $var de la fonction incremente.

5.3 Les littéraux

Chaines de caractères

Les chaines de caractères peuvent être définies soit en les encadrant avec des simples quotes (caractère ') ou avec des guillemets ("). Le comportement, dans chacun des cas, est différent.

```
PS C:\Windows\system32> $a = 'bonjour'
PS C:\Windows\system32> $a
bonjour

PS C:\Windows\system32> $b = '$a le monde'
PS C:\Windows\system32> $b
$a le monde

PS C:\Windows\system32> $b = "$a le monde"
PS C:\Windows\system32> $b
bonjour le monde
```

Ainsi qu'on peut le voir dans cet exemple, lorsque la chaine est encadrée par des guillemets, il y a substitution automatique des variables qui sont incluses dedans. Ici, la première déclaration de la variable $b ne provoque pas le remplacement de la variable $a par son contenu car le literal est encadré par des simples quotes. La deuxième déclaration provoque la substitution de la variable $a par sa valeur.

Substitution de la valeur d'une propriété d'un objet.

Ceci est très important. Pour substituer la valeur d'une propriété d'un objet, il faut utiliser la syntaxe suivante : $($objet.propriété).

Par exemple :

```
PS C:\Windows\system32> $a =  get-ChildItem c:\config.sys
PS C:\Windows\system32> "taille du fichier : $a.length octets"
taille du fichier : C:\config.sys.length octets

PS C:\Windows\system32> "taille du fichier : $($a.length) octets"
taille du fichier : 10 octets
```

La 1° ligne de cet exemple permet de récupérer les propriétés du fichier config.sys.

Dans la deuxième ligne, on essaie d'afficher la taille du fichier en utilisant une mauvaise syntaxe. Le résultat est affiché dans la 3° ligne. On n'obtient pas la taille de ce fichier.

Dans la 4° ligne , on utilise la bonne syntaxe $($a.length). Dans ce cas, la taille est bien imprimée.

Caractères d'échappement

Dans les chaines de caractères, on peut introduire un certain nombre de caractères qui vont avoir un comportement particulier. Ces caractères doivent être précédés du caractère bactick (` que l'on obtient en frappant simultanément sur altGr+7).

Caractère d'échappement	Transformation résultante
`n	Saut de ligne
`f	Saut de page
`r	Retour chariot
`a	Bip sonore
`b	Retour arrière
`t	Tabulation horizontale
`v	Tabulation verticale
`0	Null
`'	Guillemet simple
`"	Guillemet double
``	Backtick simple

5.4 Les opérateurs arithmétiques

Ce sont les mêmes qu'en Java. La priorité est aussi la même :

+	Addition
-	Soustraction
*	multiplication
/	Division
%	Modulo : calcul du reste de la division

L'opérateur + peut s'employer aussi pour concaténer des chaines de caractères./

```
PS C:\Windows\system32> $a ="Bonjour "
PS C:\Windows\system32> $b ="le monde !"
PS C:\Windows\system32> $a+$b
Bonjour le monde !

PS C:\Windows\system32> $c = "bonjour" + $b
PS C:\Windows\system32> $c
bonjourle monde !

PS C:\Windows\system32> "bonjour " + $b
bonjour le monde !
```

Les autres opérateurs mathématiques (racine carrée, élévation à la puissance, etc...) sont disponibles dans les classes .NET. Chercher sur Internet à l'aide des mots clés : .net system.math

5.5 Les opérateurs de comparaison

Les opérateurs de comparaison ont une forme bien particulière en PowerShell :

`-eq`	Égal
`-ne`	Non égal
`-gt`	Plus grand strictement
`-ge`	Plus grand ou égal
`-lt`	Plus petit strictement
`-le`	Plus petit ou égal
`-like`	Comparaison d'égalité générique (le caractère générique est * et non pas % comme en SQL)
`-notlike`	Test l'inégalité générique.
`..`	Opérateur de plage (prononcer point, point). Sert à définir une plage. Pour faire une boucle allant de 1à 10, on peut utiliser 1..10. Peut s'utiliser aussi avec des variables $a..$b signifie toutes les valeurs comprises entre $a et $b

Dans les comparaisons de chaines, les opérateurs ne prennent pas en compte la casse. Pour remédier à ce fonctionnement et rendre les comparaisons sensibles à la différenciation majuscules / minuscules, il faut faire précéder l'opérateur de la lettre « c ». Ainsi `$a -cle $b` comparera les 2 chaines de caractères en tenant compte de la casse.

5.6 Les opérateurs logiques

Ils servent à enchainer plusieurs expressions booléennes

`-and`	Et logique
`-or`	Ou logique
`-not`	Non logique
`!`	Non logique
`-xor`	Ou exclusif.

5.7 Les opérateurs d'affectation

Ils permettent d'affecter une valeur ou le résultat d'une opération à une variable :

Notation classique	***Notation raccourcie***
`$a=$a+4`	`$a+=4`
`$a=$a-4`	`$a-=4`
`$a=$a*4`	`$a*=4`
`$a=$a/4`	`$a/=4`
`$a=$a%8`	`$a%=8`
`$a=$a+1`	`$a++`
`$a=$a-1`	`$a--`

5.8 Exemple de création de boucle

Cet exemple initialise une variable et multiplie la valeur de cette variable par la valeur de a qui varie de 1 à

10. La valeur résultant est affectée à la variable.

```
PS C:\Windows\system32> $var=1
PS C:\Windows\system32> foreach ($a in 1..10) {$var*=$a; $var}
1
2
6
24
120
720
5040
40320
362880
3628800
```

6 Les structures de contrôle

6.1 Les blocs d'instruction

Comme en java, les blocs d'instructions sont encadrés par des accolades.

Un bloc peut contenir d'autres blocs.

6.2 L'instruction conditionnelle

Elle permet, selon une condition, d'exécuter ou non un bloc d'instructions :

```
$var = read-host 'Saisissez un caractère '
if ($var -eq 'A')
{
    'La valeur saisie est A'
}
```

Elle peut aussi permettre d'exécuter au choix un bloc d'instructions ou un autre

```
$nb1 = read-host 'Saisissez un 1° nombre '
$nb2 = read-host 'Saisissez un 2° nombre '
if ($nb1 -gt $nb2)
{
    $diff = $nb1 - $nb2
}
else
{
    $diff = $nb2 - $nb1
}
'Valeur absolue de la différence : ' + $diff
```

6.3 La boucle while

Elle permet de répéter un bloc d'instruction tant qu'une condition donnée est vraie.

```
$nombre = 0
$tab = 0..99       # initialisation du tableau
while ($nombre -lt $tab.length)
{
    write-host $tab[$nombre]
    $nombre++
}
```

Dans cet exemple, on imprime le contenu du tableau $tab. Remarquer l'initialisation du tableau à l'aide de l'opérateur de plage. Remarquer aussi que l'indexation se fait par le biais d'une expression entre crochets []. Comme en java, l'index va de 0 à longueur du tableau -1.

6.4 La boucle do

Elle se présente comme la boucle while, à la différence que la condition est testée après que le bloc d'instruction ait été exécuté au moins une fois.

```
do
{
    [int] $b = read-host 'Entrez une valeur comprise entre 0 et 10'
```

```
} while (($b -lt 0) -or ($b -gt 10))
'Fin de la saisie'
```

Dans cet exemple, il est demandé à l'opérateur de saisir une valeur entre 1 et 10. Le programme pourra alors continuer. Sinon, le programme bouclera jusqu'à satisfaction.

6.5 La boucle for

```
$tab = 0..99
for ($i=0; $i -lt 99; $i++)
{
    write-host $tab[$i]
}
```

Le fonctionnement est le suivant :

1. l'expression initiale est évaluée
2. L'expression conditionnelle est évaluée. Si elle est fausse, la boucle For se termine.
3. le bloc d'instruction de la boucle est exécuté.
4. L'incrémentation est réalisée.
5. On revient au point 2.

6.6 La boucle forEach

Elle permet de parcourir des listes d'objets.

```
forEach ($element in Get-Process)
{
    write-host "$($element.name) - $($element.startTime)"
}
```

Cette boucle permet de faire la liste des noms et dates de démarrage des processus en cours de fonctionnement

On peut utiliser la boucle ForEach en sortie d'un pipe de la manière suivante :

```
get-process | forEach {$_.name+' - '+$_.StartTime}
```

Cette instruction donne exactement le même résultat que le bloc précédent. Elle peut s'utiliser directement en ligne de commande.

7 Les fonctions

Les fonctions sont l'équivalent des méthodes en Java. Elles permettent de donner un nom à un ensemble d'instructions. On peut ainsi utiliser plusieurs fois la même suite d'instructions sans avoir à les réécrire, mais,

simplement en activant la fonction correspondante.

7.1 Nommage et déclaration

Une fonction se déclare de la manière suivante

```
Function [portee:] <nom de la fonction>
{
      param (<liste des paramètres>)
      # bloc d'instructions
}
```

Les deux mots clés sont « Function » qui indique qu'on commence la description d'une fonction et « param » qui indique que débute une liste de paramètre.

Exemple de fonction :

```
function print-accueil
{
    $date = get-date
    write-host "Bonjour, nous sommes le $date"
}

print-accueil
```

Voici le résultat de ce script :

```
Bonjour, nous sommes le 05/19/2012 17:01:04
```

Remarquer que le nom de la fonction n'est pas précédé du caractère $.

7.2 paramètres

Les paramètres sont des valeurs qui sont transmises à la fonction et que celle-ci va utiliser pour réaliser les opérations dont elle est chargée :

Exemple :

```
function Set-Popup
{
    param ($titre="Erreur", $message="Bonjour le monde")
    $WshShell = New-Object -ComObject wscript.shell
    $WshShell.popup($message,0,$titre)
}
```

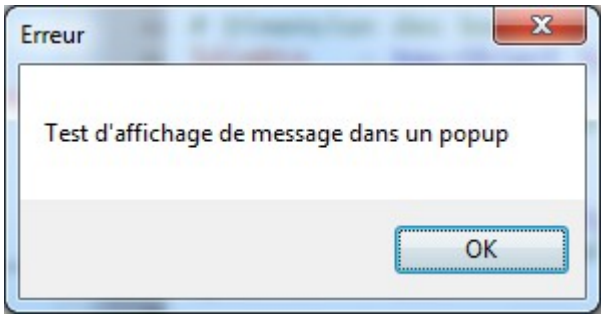
Cette fonction instancie un objet de l'OS appelé wscript.shell et active la méthode popup de cet objet qui affiche une fenêtre avec un titre et un message.

Pour activer cette fonction et lui demander d'afficher le message « Test d'affichage de message dans un popup ». on va faire appel à cette fonction de la manière suivante :

```
set-popup -message "Test d'affichage de message dans un popup"
```

Il suffit donc d'indiquer quel paramètre on veut remplir, lui donner sa valeur et appeler la fonction. On obtient

le résultat suivant :



Si on ne lui indique aucun paramètres, la fonction utilisera ici les valeurs par défaut données dans sa déclaration.

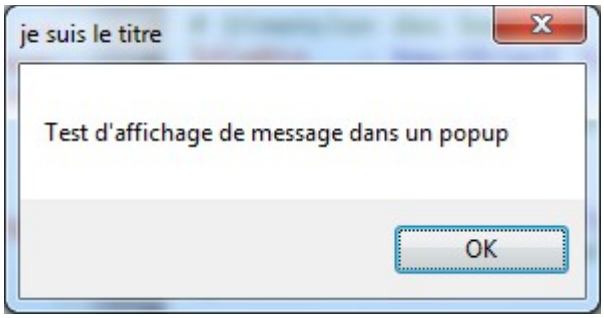
```
set-popup
```

Affiche la fenêtre suivante :

On peut aussi lui indiquer les valeurs à utiliser, sans préciser le nom des paramètres. Dans ce cas, Powershell repère les paramètres en fonction de leur position

```
set-popup "je suis le titre" "Test d'affichage de message dans un popup"
```

Cette instruction donnera le résultat suivant :



7.3 Retourner une valeur

Une fonction retourne tout objet qui est émis. Il suffit donc d'insérer l'objet à transmettre en fin de fonction ou de script pour que son résultat soit transmis à l'activateur de la fonction :

```
Function moyenne
{
      param ([double]$ n1, [double] $n2]
      ($n1 + $n2)  / 2
}
```

Pour utiliser cette fonction, il suffit de l'activer en lui transmettant les valeurs sur la quelle elle doit faire le calcul :

```
moyenne 16 10
```

Mais il est aussi possible d'écrire :

```
Function moyenne
{
      param ([double]$ n1, [double] $n2]
      return ($n1 + $n2)  / 2
}
```

8 Powershell et .Net

8.1 Utilisation des classes .NET

Dans PowerShell, il est possible d'utiliser directement des classes .NET. Pour cela, on va créer des objets à partir de ces classes. Il est donc nécessaire de connaître les différentes classes. Pour cela, il faut se rendre sur Internet sur le site : http://msdn.microsoft.com/fr-fr/library/gg145045

Par exemple, on peut utiliser des fonctions de classes qui sont prédéfinies. Ainsi pour la classe SYSTEM.MATH :

```
[system.math]::round(45.958687,2)
```

Donnera le résultat : 45,96

Pour utiliser une classe, une fonction ou un objet, il suffit dont de préciser l'espace de noms auquel il appartient, suivi de :: et du nom de la classe de l'objet ou de la constante déterminée.

On peut tenter un essai en utilisant la classe SYSTEM.DATETIME

Créons un objet correspondant à cette classe. On peut procéder de 2 manières suivantes :

```
$var = [system.dateTime]'05/19/2012'
$vb = new-object -typeName DateTime
$vb='05/19/2012'
```

A noter que .NET, pour initialiser la date, utilise le format neutre US (mm/jj/aaaa).

Il est alors possible d'utiliser les méthodes spécifiques au type DateTime de .NET. Par exemple, pour formater la date en utilisant les règles régionales :

```
[dateTime]$vb
```

Pour connaître la liste des méthodes et des propriétés de l'objet ,on peut faire appel à la commande get-member :

```
$var | get-member
```

Ainsi, pour récupérer la liste des mois, en utilisant les paramètres régionaux, on peut procéder de la manière suivante :

```
0..11 |
ForEach-Object
{ [Globalization.DatetimeFormatInfo]::CurrentInfo.MonthNames[$_] }
```

8.2 Les assemblies

Les classes .NET sont très nombreuses. Elles sont regroupées par espace de noms et stockées dans des bibliothèques spécifiques, un peu comme en Java.

Au démarrage de Powershell, un certain nombre de ces bibliothèques (appelées des assemblies) sont préchargées. Pour les connaître, on peut frapper la commande :

```
[system.appdomain]::currentdomain.getAssemblies()
```

Chaque objet de la liste retournée est du type *System.reflection.assembly* et dispose donc d'une palette de méthodes, dont la méthode getTypes qui nous donnera la liste des types qui sont disponibles dans cet assembly, c'est à dire la liste des classes disponibles.

```
# charge la liste dans un tableau $ass
$ass = [system.appdomain]::currentdomain.getAssemblies()

$ass[0] | get-members    # examen du premier élement du tableau.
```

Pour charger des assemblies supplémentaires, il faut utiliser la méthode loadWithPartialName de la classe *System.reflection.assembly* . D'autres méthodes permettent de réaliser cette opération, mais celle-ci permet d'utiliser le nom court der l'assembly et non pas le nom complet (ou nom « fort ») qui est beaucoup plus complexe à retenir.

Exemple :

```
[void] [system.reflection.Assembly]::LoadWithPartialName("MySql.Data")
```

Dans cet exemple, on instancie un objet sans nom (void) et on applique la méthode LoadWithPartialName pour charger le connecteur MySql.