2

Le langage VBScript

2. Le langage VBScript

VBScript est un des langages natifs à *Windows Script Host* avec le JScript. Le langage VBScript est moins stricte que son homologue et demeure donc plus accessible que le JScript. Le langage VBScript tire ses racines du Visual Basic et en emprunte donc la majorité des structures et des éléments syntaxiques.

Règles fondamentales

D'abord, sachez que le VBScript **n'est pas discriminant à la casse** et que, conséquemment, vous pouvez alterner les minuscules et les majuscules au sein de votre code sans aucune différence pour le moteur de script. Ainsi, les trois instructions suivantes sont équivalentes :

```
WScript.Echo
wscript.echo
WSCRIPT.ECHO
```

Cependant, il est déconseillé d'écrire l'ensemble du code complètement en minuscules ou complètement en majuscules puisque cette pratique produit des codes difficilement lisibles.

Ensuite, VBScript **ne tient pas compte des espaces supplémentaires** qui pourraient être insérées au sein du code. Ainsi, les deux instructions suivantes son équivalentes :

```
WScript.Echo "Allô la planète"
WScript.               Echo         "Allô la planète"
```

VBScript n'impose aucun maximum quant au nombre de caractères sur une même ligne tel que d'autres langages tels que Visual Basic lui-même. Cependant, il peut être intéressant de contenir le code en largeur pour des fins de lisibilité. VBScript vous permet donc de briser une ligne de code et de la **continuer sur la ligne suivante en insérant un caractère de continuation** composé d'un espace suivi d'un trait de soulignement (_) :

```
WScript.Echo "Message à " & vUtilisateur _
& " Veuillez vous assurer de verrouiller ou " _
&  " de fermer votre station avant de quitter."
```

Par ailleurs, il vous est possible d'inscrire plusieurs instructions sur une même ligne en les séparant par un symbole deux-points (:). Ainsi, la ligne suivante :

```
WScript.Echo "Bonjour"  :  WScript.Echo "Bonsoir"
```

est équivalente aux lignes suivantes :

```
WScript.Echo "Bonjour"
WScript.Echo "Bonsoir"
```

Cependant, cette pratique tend grandement à diminuer la lisibilité du code.

L'utilisation de variables

Les variables permettent au programmeur d'accéder à la mémoire de l'ordinateur afin d'y stocker et d'y récupérer des données. Une variable possède un nom agissant en tant qu'alias vers une adresse-mémoire donnée et le programmeur peut accéder au contenu de cette adresse-mémoire tout simplement en utilisant le nom de la variable.

```
MaVariable = 15                    ' Stocke la valeur 15 dans la variable

WScript.Echo MaVariable            ' Affiche le contenu de la variable : 15

MaVariable = MaVariable + 10       ' Stocke le résultat de l'addition 15 + 10

WScript.Echo MaVariable            ' Affiche le contenu de la variable : 25
```

Vous remarquez certainement que le contenu d'une variable peut changer au cours de l'exécution du script. Cependant, une variable ne peut contenir qu'une seule donnée à la fois. On peut parfois nommer ces variables *« scalaires »* en oppositions aux variables *« vectorielles »* pouvant contenir plusieurs valeurs que la programmation informatique nomme *« tableaux »*. Il sera question des tableaux plus loin.

Les variables peuvent manipuler des données de différents types.

- Les variables **numériques** peuvent contenir des valeurs numériques entières ou réelles sur lesquelles diverses opérations mathématiques peuvent être effectuées.

```
nChiffreA  = 5
nChiffreB = 10
nChiffreC = (nChiffreA * 2) + nChiffreB
```

- Les variables **chaînes de caractères** peuvent contenir une suite de caractères qui, mis ensembles, forment un mot, une phrase ou une expression. Certaines opérations typiques aux chaînes de caractères peuvent être effectuées sur ces variables. Une nouvelle valeur est attribuée à ce type de variables en incluant les caractères entre deux guillemets *(" ")* :

```
strNom = "Pinchaud"
strVoiture = "Ford Mustang"
WScript.Echo strNom & " conduit une voiture " & strVoiture
```

- Les variables **dates** peuvent contenir des dates. Certaines opérations typiques aux dates peuvent être effectuées sur ces variables. Une nouvelle valeur est attribuée à ce type de variables en incluant la date entre deux dièses *(# #)*:

```
dAujourdhui = #15-03-2002#
```

La déclaration de variables

La déclaration d'une variable est l'acte par laquelle le programme spécifie au script la liste des variables qui seront utilisées et que le script doit s'attendre à rencontre au sein des différentes instructions composants ce script.

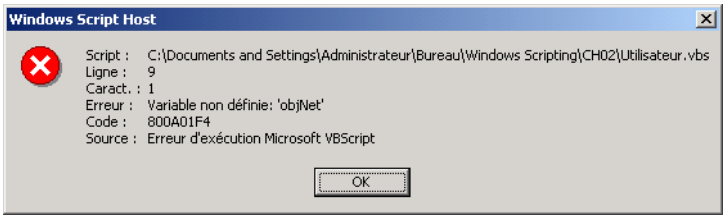
Quoique la majorité des langages de programmation requièrent la déclaration des variables avant que celles-ci ne puissent être utilisées au sein du code, VBScript n'impose pas cette restriction. VBScript considère qu'une variable est déclarée dès la première fois qu'elle est référencée au sein du code. Brièvement, dès que VBScript rencontre une expression qui n'est ni le nom d'une procédure ni un mot-clé réservé par le langage, il considère qu'il s'agit conséquemment d'une variable et procède automatiquement à sa déclaration. Ne voyez pas ce comportement comme une échappatoire vers la simplicité mais plutôt comme une source fréquente d'erreurs difficiles à diagnostiquer. Dans l'exemple suivant, le programmeur a effectué une faute de frappe. Le script ne génère aucune erreur mais ne produit pas le résultat attendu :

```
nMonAge = 30
nMonAge = nMoAge + 10
WScript.Echo nMonAge
```

Le script affiche 10 alors que, à première vue, il devrait afficher le résultat de l'addition de 30 + 10 et conséquemment afficher 40. Cependant, sur la deuxième ligne de code, le programmeur a nommé la variable *nMoAge* au lieu de *nMonAge*. VBScript y a vu une nouvelle expression et a déclaré une seconde variable qu'il a initialisé à zéro. Ainsi, la variable *nMonAge* se fait assigner le résultat de l'addition de 0 + 10. Puisque ce type d'erreurs est difficile à diagnostiquer, il nous sera préférable de demander à VBScript de modifier son comportement par défaut en nous obligeant à déclarer dûment nos variables en introduisant la directive `Option Explicit` en tant que toute première instruction au sein de nos scripts :

```
Option Explicit
```

Ainsi, **chacune des variables devra préalablement être déclarée avant d'être utilisée** au sein du code. Toute faute de frappe interromprait l'exécution du script et provoquerait le soulèvement d'une erreur `#800A01F4` (`'Variable non définie'`) par VBScript comme suit :



Il ne s'agit pas ici de chercher à provoquer inutilement des erreurs mais plutôt de se donner les outils nécessaires afin d'être en mesure de les détecter afin d'obtenir un script efficace produisant les résultats réellement attendus.

La déclaration d'une variable peut s'effectuer à l'aide d'un des quatre mots-clé réservés suivants :

- **Dim** provoque la déclaration d'une variable privée.
- **Private** provoque la déclaration d'une variable privée.
- **Public** provoque la déclaration d'une variable publique.
- **Static** provoque la déclaration d'une variable statique interne à la procédure.

Nous éclaircirons plus loin au sein de ce chapitre la signification de chacun de ces mots-clé mais sachez pour l'instant que vous pouvez utiliser l'un ou l'autre de ces mots-clé afin de déclarer adéquatement une variable comme dans le script suivant :

```
Option Explicit

Dim objNet

Set objNet = CreateObject("WScript.Network")
objNet.MapNetworkDrive "Z:", "\\serveur\partage"
WScript.Echo "Le lecteur Z: est connecté!"
```

CH02\map.vbs

Finalement, le nom d'une variable **doit débuter par un caractère alphabétique** *(a-z, A-Z)*, peut subséquemment contenir des caractères alphanumériques mais ne peut posséder un point (.) et **ne peut excéder 255 caractères** en longueur.

Quoiqu'aucune autre règle formelle autre que celles énumérées précédemment ne soit prévue par VBScript au sujet de la nomination des variables, certaines conventions établies par la maturité de la programmation informatique proposent certaines pratiques permettant d'augmenter la lisibilité du code. Une de ces conventions stipule qu'il peut être bon de réserver certains des premiers caractères composants le nom d'une variable afin d'identifier le type de données qu'y sera stocké. Le tableau présente certains exemples mettant en œuvre cette convention :

	Type de variable	Exemple
`n`	Valeur numérique *(tous types confondus)*	`Dim nCompteur`
`b`	Valeur booléenne	`Dim bReponse`
`d`	Date	`Dim dAujourdhui`
`obj`	Variable-objet	`Dim objNet`
`str`	Chaîne de caractères	`Dim strAdresse`

Déclaration et utilisation de constantes

Une constante est une variable à laquelle une valeur est assignée au début du code et qui ne peut être modifiée et qui doit demeurer inchangée tout au long de l'exécution du script. L'utilisation de constantes permet simplement d'augmenter la lisibilité de votre code et d'en simplifier efficacement la compréhension et la gestion. Donc, au lieu d'utiliser un code comme le suivant :

```
MaVariableA  = 3.1416 * 2
MaVariableB  = 3.1416 * 4
```

La déclaration d'une constante à l'aide du mot-clé `Const` en augmenterait la compréhension :

```
Const PI = 3.1416
MaVariableA = PI * 2
MaVariableB = PI * 4
```

si, lors de la conception de votre code, vous aviez à modifier la totalité des endroits où la valeur de PI est utilisée, vous n'auriez qu'à modifier la valeur de la constante pour que les modifications ne se répercutent dans l'ensemble du script.

```
Const PI = 3.1415923
MaVariableA = PI * 2
MaVariableB = PI * 4
```

Utilisation de constantes existantes

Les différents modèles d'objets que vous apprendrez à manipuler prévoient une pluralité de constantes que vous pouvez utiliser au sein de vos scripts. Vous pouvez connaître ces constantes en consultant la documentation appropriée ou en consultant un navigateur d'objets tel celui fourni avec *Microsoft Visual Basic* ou *Microsoft Access*. Vous apprendrez à utiliser un navigateur d'objets à la fin du présent chapitre.

Le langage *VBScript* prévoit également plusieurs constantes permettant de manipuler plus aisément les différents éléments du langage. Notez que la valeur des constantes peut être déclarée en valeur hexadécimale en précédant la valeur de la constante par le préfixe `&H`. Ainsi, une constante déclarée comme possédant la valeur `&H20` vaut réellement `32` en décimal.

Par exemple, l'instruction suivante :

```
WScript.Echo "Bonjour " & vbCrLf & "Nous sommes le " & Date()
```

L'exemple ci-haut affiche le résultat ci-contre. Notez l'utilisation de la constante `vbCrLf` forçant le saut de ligne entre les deux expressions.

La liste complète des constantes intrinsèques à *VBScript* vous seront présentées au fil du présent chapitre.

Types de données et conversion explicite

Un type de données s'applique généralement à une variable et défini quel type d'informations pourront être stockés par cette variable. Par exemple, une variable pourrait pouvoir contenir des valeurs numériques entières tandis qu'une autre variable pourrait pouvoir contenir des chaînes de caractères. L'utilisation du type de données approprié peut devenir importante lorsque, par exemple, le contenu des variables `X` et `Y` sont additionnés afin d'en récupérer le résultat :

```
X = 4
Y = 2
Z = X + Y          'Z vaut 6
```

Ainsi, lorsque les variables `X` et `Y` sont de type nombre entier, `Z` prendra la valeur 6, résultat de l'addition mathématique des deux entiers. Cependant, considérez l'exemple suivant :

```
X = "4"
Y = "2"
Z = X + Y          'Z vaut "42"
```

Lorsque les variables `X` et `Y` sont de type caractère, `Z` prendra la valeur "42", résultat de la concaténation des deux chaînes de caractères mises bout à bout.

Contrairement à la majorité des langages structurés, *VBScript* ne reconnaît qu'un seul type de données pour dimensionner les variables : le type de données **variant**. Différentes fonctions de conversion explicite permettent, par exemple, de convertir des chaînes de caractères en valeurs numériques entières afin d'appliquer une addition mathématique adéquate sur ces valeurs :

```
X = "4"
Y = "2"
Z = CInt(X) + CInt(Y)          'Z vaut 6
```

Lorsque la valeur des variables `X` et `Y` sont explicitement converties en valeurs numériques entières à l'aide de la fonction `CInt`, `Z` prendra la valeur 6, résultat de l'addition mathématique des deux chaînes de caractères préalablement converties en entiers.

Fonction	Description
CBool	Retourne une valeur booléenne (`True|False`).
CByte	Retourne un entier contenu entre 0 et 255.
CCur	Retourne un réel contenu entre –922 337 203 685 477.5808 et +922 337 203 685 477.5807.
CDate	Retourne une date valide.
CDbl	Retourne un réel contenu entre –1.797 693 134 862 32 E308 et –4.940 656 458 412 47E-324 pour les valeurs négatives et entre 4.940 656 458 412 47E-324 et 1.797 693 134 862 32 E308 pour les valeurs positives.
CInt	Retourne un entier contenu entre –32 768 et +32 767.
CLng	Retourne un entier contenu entre –2 147 483 648 et +2 147 483 647.
CSng	Retourne un réel contenu entre –3.402 823 E38 et –1.401 298 E-45 pour les valeurs négatives et entre 1.401 298 E-45 et 3.402 823 E38 pour les valeurs positives.
CStr	Retourne une chaîne de caractères.

Les opérateurs

Les opérateurs sont des symboles réservés par le langage *VBScript* permettant une pluralité d'opérations arithmétiques, logiques, comparatives et autres. Ainsi, vous ne pouvez vous servir de ces symboles pour d'autres fins que celles prescrites par le langage.

Opérateurs arithmétiques
Les opérateurs arithmétiques permettent d'effectuer des opérations arithmétiques telles l'addition, la soustraction, la multiplication, etc.

Opérateur	Description	Exemple
^	Soulève un nombre à la puissance spécifiée.	`Z = 4 ^ 2` ' donne 16.
-	Précise l'inverse arithmétique d'un nombre ou d'une expression.	`Z = -(3 * 4)` ' donne –12.
*	Multiplie deux nombres.	`Z = 3 * 4`
/	Divise deux nombres en conservant les décimales le cas échéant.	`Z = 9 / 4` ' donne 2,25.
\	Divise deux nombres en tronquant les décimales le cas échéant.	`Z = 9 \ 4` ' donne 2.
Mod	Retourne le reste d'une division entière *(modulo)*.	`Z = 9 mod 4` ' donne 1.
+	Additionne deux nombres.	`Z = 4 + 3`
-	Soustrait deux nombres.	`Z = 4 – 3`

Tâchez de bien distinguer les deux opérateurs de division dont l'un deux ne considère pas les décimales le cas échéant et qui peut produire des résultats inattendus s'il est incorrectement utilisé.

Opérateur de concaténation
La concaténation est une opération qui s'applique sur des chaînes de caractères seulement. Cette opération permet d'embouter deux chaînes de caractères afin de n'en former qu'une seule.

```
strPrnom = "Joe"
strNom = "Castagnette"

WScript.Echo strPrnom & " " & strNom        ‘ Affiche "Joe Castagnette"
```

L'opérateur de concaténation ne s'appliquant que sur des chaînes de caractères, il procèdera à une conversion explicite en chaînes de caractères si des valeurs numériques lui sont fournies.

```
X = 2
Y = 6
WScript.Echo X & Y              ‘ Affiche "26"
```

Opérateur	Description	Exemple
&	Concatène deux chaînes de caractères.	`strA = strPrnom & strNom`

Opérateurs de comparaison
Ces opérateurs permettent de comparer deux valeurs. Le résultat de la comparaison s'évalue toujours par **True** ou **False**. L'utilité des opérateurs de comparaison prendra tout son sens lorsque nous aborderons les structures de contrôle plus loin dans ce chapitre.

Opérateur	Description	Exemple
=	Retourne `True` si les deux valeurs sont égales.	`bResultat = x = 5`
<>	Retourne `True` si les deux valeurs sont différentes.	`bResultat = x <> 5`
>	Retourne `True` si la valeur de gauche est plus grande que la valeur de droite.	`bResultat = x > 5`
<	Retourne `True` si la valeur de gauche est plus petite que la valeur de droite.	`bResultat = x < 5`
>=	Retourne `True` si la valeur de gauche est plus grande ou égale à la valeur de droite.	`bResultat = x >= 5`
<=	Retourne `True` si la valeur de gauche est plus petite ou égale à la valeur de droite.	`bResultat = x <= 5`
Is	Retourne `True` si les deux objets sont les mêmes.	`bResultat = objX Is objY`

Opérateurs logiques
Tout comme les opérateurs de comparaison, le résultat généré par l'utilisation des opérateurs logiques s'évalue toujours par **True** ou **False**. L'utilité des opérateurs logiques prendra tout son sens lorsque nous aborderons les structures de contrôle plus loin dans ce chapitre.

Opérateur	Description	Exemple
Not	Retourne la négation logique de l'expression.	`bR = Not (X < 1)`
And	Retourne la conjonction logique de deux expressions.	`bR = (X > 2) And (Y =1)`
Or	Retourne la disjonction logique de deux expressions.	`bR = (X > 2) Or (Y =1)`
XOr	Retourne la disjonction logique exclusive de deux expressions.	`bR = (X > 2) XOr (Y =1)`
Eqv	Retourne l'équivalence logique de deux expressions.	`bR = 2 Eqv (6 / 3)`
Imp	Retourne l'implication logique de deux expressions.	`bR = 2 Imp objNull`

Voici les tables de vérité des opérateurs logiques. Les expressions `Expr1` et `Expr2` combinées à l'aide de l'opérateur concerné donnent le résultat spécifié dans la colonne `=`.

Not	
Expr	=
Faux	Vrai
Vrai	Faux

And		
Expr1	Expr2	=
Faux	Faux	Faux
Faux	Vrai	Faux
Vrai	Faux	Faux
Vrai	Vrai	Vrai

Or		
Expr1	Expr2	=
Faux	Faux	Faux
Faux	Vrai	Vrai
Vrai	Faux	Vrai
Vrai	Vrai	Vrai

XOr		
Expr1	Expr2	=
Faux	Faux	Faux
Faux	Vrai	Vrai
Vrai	Faux	Vrai
Vrai	Vrai	Faux

Eqv		
Expr1	Expr2	=
Faux	Faux	Vrai
Faux	Vrai	Faux
Vrai	Faux	Faux
Vrai	Vrai	Vrai

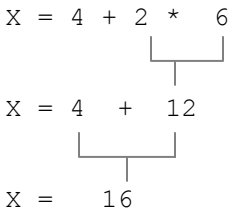
Imp		
Expr1	Expr2	=
Faux	Faux	Vrai
Vrai	Faux	Faux
Vrai	Null	Null
Faux	Vrai	Vrai
Vrai	Vrai	Vrai
Faux	Null	*Vrai*
Null	Vrai	Vrai
Null	Faux	Null
Null	Null	Null

Priorité des opérateurs

L'ensemble des opérateurs sont reliés selon un ordre de priorité très précise régissant l'ordre dans laquelle les différentes expressions doivent être évaluées par le moteur de script afin d'en calculer le résultat. Ainsi, VBScript évaluera l'expression suivante :

```
X = 4 + 2 * 6
```

et obtiendra le résultat 16 et non le résultat 36. Le secret réside dans la priorité des opérateurs qui stipule entres autres que la multiplication est prioritaire à l'addition. Voyez comment *VBScript* évalue l'expression précédente :



Afin de s'assurer d'obtenir les résultats escomptés sans trop se soucier de la priorité des opérateurs, il suffit d'insérer explicitement des parenthèses qui demeurent les opérateurs possédant la plus forte priorité :

```
X = 4 + (2 * 6)          ' Donne 16 et le code est clair !
X = (4 + 2) * 6          ' Donne 36 et le code est clair !
```

Voici le tableau complet des opérateurs VBScript présentés en ordre décroissant de priorité.

Priorité	Opérateur(s)	Description(s)
14	( )	Parenthèses explicites.
13	^	Exposant.
12	-	Inverse arithmétique.
11	* / \	Multiplication, division réelle et division entière.
10	Mod	Modulo.
9	+ -	Addition et soustraction.
8	&	Concaténation de chaînes de caractères.
7	=, <>, >, <, >=, <=, Is	Comparaisons.
6	Not	Inverse logique.
5	And	Conjonction logique.
4	Or	Disjonction logique.
3	XOr	Disjonction logique exclusive.
2	Eqv	Équivalence logique.
1	Imp	Implication logique.

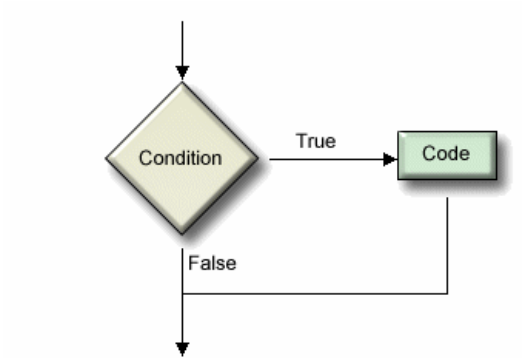
Structures de contrôle et conditions

L'exécution du code de votre script peut être contrôlé par diverses expressions appelées structures de contrôles. Les structures de contrôles sont réunies en deux groupes :

- Les **structures de branchement**, ou tests alternatifs, permettent d'exécuter un bloc de code lorsqu'une condition est rencontrée et d'exécuter un bloc de code différent lorsque la même condition n'est pas respectée.
- Les **structures répétitives**, ou itérations, permettent de répéter l'exécution d'un bloc de code tant qu'une condition est rencontrée ou qu'elle n'est pas respectée.

La structure de branchement If…Else…End If

Cette structure de branchement permet d'exécuter un bloc de code si une condition spécifiée est remplie.



Le code s'exécute seulement lorsque la condition peut positivement être évaluée par `True`. Dans tous les autres cas, aucun code supplémentaire n'est exécuté.

La syntaxe de cette structure de branchement est la suivante :

```
If <condition> Then
        <Code à exécuter>
End If
```

L'exemple suivant tire un nombre aléatoire entre 0 et 100 et en affiche la valeur. Lorsque X est un nombre pair, une boîte de dialogue le précise à l'utilisateur :

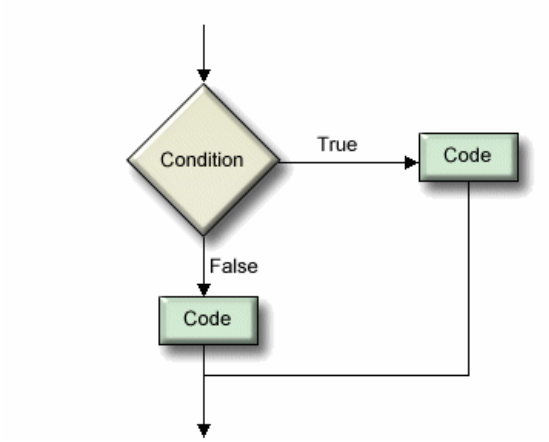
```
Randomize
X = CInt(Rnd * 100)

WScript.Echo "X vaut " & X
If (X Mod 2 = 0) Then
        WScript.Echo "X est un nombre pair."
End If
```

Il est possible d'écrire un branchement `If` sur une seule ligne lorsque le code à exécuter est constitué d'une seule instruction au lieu d'un bloc d'instructions :

```
If (X Mod 2 = 0) Then WScript.Echo "X est un nombre pair."
```

Cependant, cette syntaxe diminue la lisibilité du code et ne permet l'exécution d'une seule instruction au sein du bloc `If`. Notez que cette syntaxe ne nécessite pas l'insertion de l'instruction `End If` à la fin de la structure de branchement.

La structure peut être étendue afin d'intégrer un branchement alternatif **`Else`** dont le code ne s'exécutera que lorsque la condition ne sera pas rencontrée.



Le premier code s'exécute seulement lorsque la condition peut positivement être évaluée par `True`. Dans tous les autres cas, le second code est exécuté. Un des deux codes est nécessairement exécuté dans tous les cas et seulement un des deux codes est exécuté.

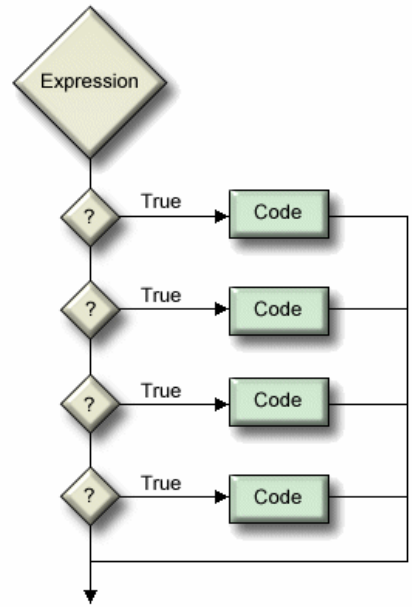
La syntaxe de cette structure de branchement est la suivante :

```
If <condition> Then

        <Code à exécuter>
Else

        <Code à exécuter>
End If
```

Voici le même exemple présenté précédemment à lequel un branchement `Else` a été ajouté :

```
Randomize
X = CInt(Rnd * 100)

WScript.Echo "X vaut " & X
If (X Mod 2 = 0) Then
        WScript.Echo "X est un nombre pair."
Else
        WScript.Echo "X est un nombre impair."
End If
```

La structure de branchement Select Case…End Select
Cette structure de branchement permet d'exécuter un bloc de code selon la valeur d'une expression spécifiée.



Un seul des blocs de code s'exécute selon la valeur de l'expression soumise à l'évaluation.

La syntaxe de cette structure de branchement est la suivante :

```
Select Case <expression>
  Case <valeur1>
       <Code à exécuter>

  Case <valeur2>
       <Code à exécuter>

  Case Else
       <Code à exécuter>

End Select
```

Si aucun des branchements ne correspond à l'expression spécifiée, le bloc facultatif `Case Else` est exécuté s'il a été spécifié.

L'exemple simple suivant montre l'utilisation de la structure `Select Case`. Notez que vous pouvez toujours insérer le nombre de blocs `Case` que désiré et que le bloc `Case Else` est facultatif.

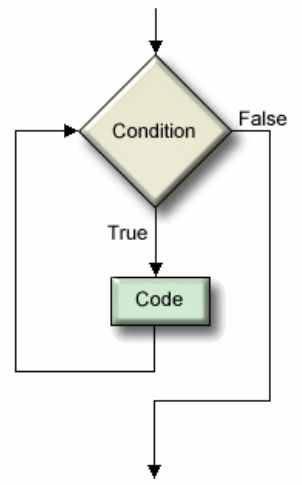
```
Select Case X
   Case 0
      WScript.Echo "X = 0"

   Case 1
      WScript.Echo "X = 1"

   Case 2
      WScript.Echo "X = 2"

   Case Else
      WScript.Echo "Désolé, la valeur est invalide!"
End Select
```

Voici un autre exemple de l'application de cette structure de branchement :

```
Select Case X
   Case 0 To 9
      WScript.Echo "X est entre 0 et 9."

   Case 10 To 99
      WScript.Echo "X est entre 10 et 99."

   Case Is > 99
      WScript.Echo "X est plus grand que 99."
End Select
```

Structures répétitives Do While/Until … Loop

Ces structures répétitives sont utilisées afin de répéter une instruction ou un certain bloc de code.



Le code est exécuté tant que la condition est vraie *(boucle while)* ou jusqu'à ce que la condition soit vraie *(boucle Until)*. Lorsque la condition n'est plus remplie *(boucle while)* ou lorsqu'elle est remplie *(boucle Until)*, l'exécution quitte la boucle afin de poursuivre normalement le code suivant.

Les syntaxes de ces structures répétitives sont les suivantes :

```
Do While <condition>
      <Code à exécuter>
Loop


Do Until <condition>
      <Code à exécuter>
Loop
```

Voici un exemple simple de la mise en œuvre de ces structures répétitives. La boîte de message est affichée 10 fois en affichant chacune des valeurs entre 0 et 9 inclusivement :

```
nNbr = 0
Do While nNbr < 10
      WScript.Echo "nNbr vaut " & nNbr
      nNbr = nNbr + 1
Loop
```

Le même résultat peut être obtenu à l'aide de la boucle `Do Until` à la condition de modifier la condition :

```
nNbr = 0
Do Until nNbr = 10
      WScript.Echo "nNbr vaut " & nNbr
      nNbr = nNbr + 1
Loop
```

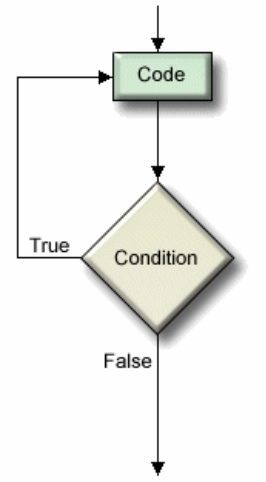
Il est possible de quitter prématurément une boucle à l'aide de l'instruction **Exit Do**. Lorsque cette instruction est rencontrée, la structure répétitive est quittée peu importe l'état de la condition gérant la boucle.

L'exemple suivant génère le même résultat que les deux boucles présentées précédemment :

```
nNbr = 0
Do
      WScript.Echo "nNbr vaut " & nNbr
      nNbr = nNbr + 1

      If nNbr >= 10 Then Exit Do
Loop
```

Structures répétitives Do … Loop While/Until

Comme les structures présentées précédemment, ces structures répétitives sont utilisées afin de répéter une instruction ou un certain bloc de code.



Le code est exécuté tant que la condition est vraie *(boucle while)* ou jusqu'à ce que la condition soit vraie *(boucle Until)*. Lorsque la condition n'est plus remplie *(boucle while)* ou lorsqu'elle est remplie *(boucle Until)*, l'exécution quitte la boucle afin de poursuivre normalement le code suivant. Notez que, contrairement aux précédentes boucles `Do`, les présentes boucles exécutent le bloc de code au moins une fois puisque la condition est évaluée après que celui-ci ne soit exécuté.

Les syntaxes de ces structures répétitives sont les suivantes :

```
Do
      <Code à exécuter>
Loop While <condition>
```

```
Do
      <Code à exécuter>
Loop Until <condition>
```

Voici un exemple simple de la mise en œuvre de ces structures répétitives. La boîte de message est affichée 10 fois en affichant chacune des valeurs entre 0 et 9 inclusivement :

```
nNbr = 0
Do
      WScript.Echo "nNbr vaut " & nNbr
      nNbr = nNbr + 1
Loop While nNbr < 10
```

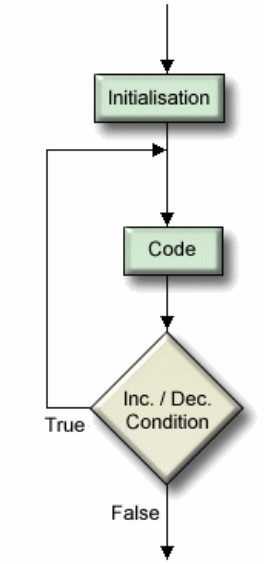
Le même résultat peut être obtenu à l'aide de la boucle `Do Until` à la condition de modifier la condition :

```
nNbr = 0
Do
      WScript.Echo "nNbr vaut " & nNbr
      nNbr = nNbr + 1
Loop Until nNbr = 10
```

Ces structures répétitives acceptent également l'utilisation de l'instruction **Exit Do**.

Structure répétitive For… Next

Comme les structures présentées précédemment, cette structure répétitive est utilisée afin de répéter une instruction ou un certain bloc de code. La boucle `For… Next` présente cependant l'avantage d'inclure un compteur permettant d'initialiser le nombre d'itérations à exécuter.



Le compteur est d'abord initialisé ainsi que la valeur maximale ou minimale qu'il doit atteindre. Le code est ensuite répété tant et aussi longtemps que le compteur n'a pas atteint ou dépassé la valeur prescrite lors de l'initialisation de la boucle. Le compteur est automatiquement incrémenté ou décrémenté après l'exécution du code.

La syntaxe de cette structure répétitive est la suivante :

```
For <compteur> = <Valeur1> To <Valeur2>
      <Code à exécuter>
Next
```

La variable servant de compteur doit préalablement être déclarée si votre code prévoit la déclaration explicite des variables à l'aide de la directive `Option Explicit` expliquée précédemment au début de ce chapitre.

Voici un exemple simple de la mise en œuvre de cette structure répétitive. La boîte de message est affichée 10 fois en affichant chacune des valeurs entre 0 et 9 inclusivement :

```
For nCpt = 0 To 9
      WScript.Echo "nCpt vaut " & nCpt
Next
```

Il est possible de créer des boucles inversées au sein desquelles la valeur du compteur est automatiquement décrémentée au lieu d'être incrémentée *(par défaut)* en spécifiant une valeur négative à l'instruction facultative **Step**. L'exemple suivant affiche les valeurs entre 0 et 9 inclusivement mais débute en affichant 9 avant de terminer par 0.

```
For nCpt = 9 To 0 Step -1
      WScript.Echo "nCpt vaut " & nCpt
Next
```

De la même manière il est possible de créer des boucles dont l'incrémentation ou la décrémentation du compteur s'effectue par multiple de 2, de 3 ou de toute autre valeur. L'exemple suivant affiche les nombres pairs compris entre 0 et 20 inclusivement :

```
For nCpt = 0 To 20 Step 2
      WScript.Echo nCpt & " est un nombre pair"
Next
```

Il est possible de quitter prématurément la boucle `For… Next` en introduisant l'instruction **Exit For** au sein du code à exécuter.

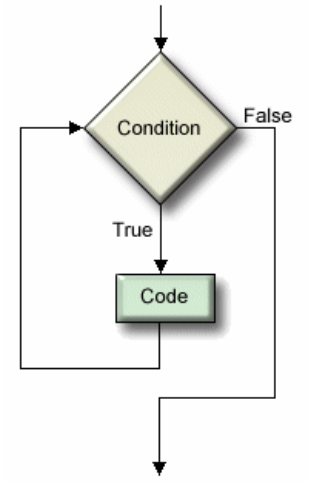
Structure répétitive For Each… Next

Comme les structures présentées précédemment, cette structure répétitive est utilisée afin de répéter une instruction ou un certain bloc de code mais utilise le nombre d'objets membres d'une collection afin d'initialiser le nombre le nombre d'itérations à effectuer. Ces concepts orientés objets vous seront exposés plus tard.

```
For Each File in objFiles
      WScript.Echo  File.Name
Next
```

Structure répétitive While… Wend

Comme les structures présentées précédemment, cette structure répétitive est utilisée afin de répéter une instruction ou un certain bloc de code. Cependant, cette structure existe depuis les débuts du langage *Basic* et est supplantée par les autres formes de boucles. La boucle `While…Wend` a été conservée pour des fins de compatibilité antérieure mais il est conseillé d'utiliser les autres structures répétitives mieux structurées et plus flexibles.



Le code est exécuté tant que la condition est vraie. Lorsque la condition n'est plus remplie, l'exécution quitte la boucle afin de poursuivre normalement le code suivant.

La syntaxe de cette structure répétitive est la suivante :

```
While <condition>
      <Code à exécuter>
Wend
```

Voici un exemple simple de la mise en œuvre de cette structure répétitive. La boîte de message est affichée 10 fois en affichant chacune des valeurs entre 0 et 9 inclusivement :

```
nNbr = 0
While nNbr < 10
      WScript.Echo "nNbr vaut " & nNbr
      nNbr = nNbr + 1
Wend
```

Les programmeurs qui auraient utilisé Visual Basic noteront que VBScript ne supporte pas certaines structures de contrôle de ce langage telles que `IIf`, `Choose` et `Switch`.

Focus sur la condition

La condition qui régi l'exécution ou non d'un bloc de code au sein d'une structure de contrôle est toujours évaluée par une valeur booléenne `True` ou `False`. VBScript réduira toujours à ce niveau la valeur de la condition que votre code lui soumet. Dans l'exemple suivant, la condition est simplement composée d'une valeur numérique :

```
nVar = 14
Do While nVar
      MsgBox "nVar vaut " & nVar
      nVar = nVar - 1
Loop
```

Or, ce script affiche les différentes valeurs de la variable `nVar` de 14 à 1 puis le code cesse de s'exécuter. Pourquoi ? Ce comportement s'explique aisément par la façon qu'ont les langages de programmation d'évaluer les conditions selon l'algorithme booléen qui stipule que :

- Une **valeur nulle** (0) s'évalue négativement et prend la valeur **False**.
- **Toute autre valeur** s'évalue positivement et prend la valeur **True**.

Ainsi, l'exemple précédent évalue positivement la condition de la boucle tant que la variable `nVar` possède une valeur différente de zéro.

Il existe également quelques pièges qu'il faut faire attention d'éviter lors de la construction de boucles et de conditions. Un d'eux provient souvent de la méprise du programmeur à l'égard des opérateurs logiques AND et OR. Dans l'exemple suivant, le script attend une réponse de l'utilisateur avant de poursuivre l'exécution. À l'aide d'une boucle, le script s'assure que l'utilisateur ne puisse entrer que deux réponses possibles… ou, du moins, il croit le faire :

```
'********* Ce code est erroné… ne l'essayez-pas **********'
Do
      Reponse = InputBox("Désirez-vous continuer ? (Oui, Non)")
Loop While LCase(Reponse) <> "oui" OR LCase(Reponse) <> "non"
```

Cependant, la table de vérité de l'opérateur OR stipule que l'une des deux parties de l'expression évaluée doit être vraie pour que l'expression s'évalue positivement. Ainsi, si l'utilisateur entre la réponse *"non"*, le script évalue d'abord `Reponse <> "oui"` et conclu, en effet que cette affirmation est vraie et, concluant avec raison que la condition est vraie, procède à une nouvelle itération de la boucle. Le code aurait donc dû se lire comme suit :

```
Do
      Reponse = InputBox("Désirez-vous continuer ? (Oui, Non)")
Loop While LCase(Reponse) <> "oui" AND LCase(Reponse) <> "non"
```

Un autre piège provient des conditions mal construites provoquant des boucles infinies. Examinez l'exemple suivant et notez que la condition de cette boucle ne pourra jamais être rencontrée.

```
'********* Ce code est erroné… ne l'essayez-pas ***************'
nVar = 0
Do Until nVar > 10
      Y = nVar * X
Loop
```

Cet exemple illustre bien le phénomène des boucles infinies. La condition ne pouvant jamais être rencontrée, le script s'exécute éternellement et fait rapidement grimper à 100% le niveau d'utilisation du processeur. Notez que *Windows Script Host* ne possède qu'une seule technique afin de stopper les boucles infinies lorsque vous exécutez le script à l'extérieur du débogueur : appuyez sur les touches `CTRL + ALT + DEL` afin d'obtenir le *Gestionnaire de tâches* et localisez l'application et le processus **CScript.exe** ou **WScript.exe** afin de les stopper.

Imbrication de structures de contrôles

VBScript permet au programmeur d'imbriquer les différentes structures de contrôles afin de donner plus de latitude à son code. L'imbrication de structures de contrôles emploie le principe dit LIFO *(Last-In, First-Out)* stipulant que la dernière structure de contrôle imbriquée doit être la première à se terminer. Voici l'exemple d'un code permettant de connaître tous les nombres premiers compris entre 1 et 100 :

```
For X = 1 To 100
      Premier = True

    For Y = 2 To X

          If (X Mod Y = 0) AND (X <> Y) Then
                Premier = False
                Exit For
          End If

    Next

      If Premier Then WScript.Echo X & " est un nombre premier."
Next
```

CH02\NbrPremiers.vbs

Notez que chacune des structures de contrôle se termine avant que ne se termine la structure de contrôle parent.

Utilisation des fonctions

VBScript hérite de ses ancêtres *Basic* et *Visual Basic* d'une multitude de fonctions procurant une grande flexibilité à ce langage. Ces fonctions sont intrinsèques au langage et ne nécessite donc pas que le script référence un modèle d'objets externe tel que vous apprendrez à les utiliser dans les prochains chapitres. Lorsque votre script invoque une fonction, votre script demande au code qui y est contenu de s'exécuter.

Il est possible d'invoquer une fonction ou une procédure simplement en la nommant:

```
NomFonction
```

Cet appel d'une fonction met en oeuvre la syntaxe la plus simple puisque la fonction ne nécessite pas que des paramètres d'entrée ne lui soient spécifiés. Les paramètres représentent des informations supplémentaires que la fonctions pourrait attendre afin de préciser certains aspects de l'appel. Ainsi, lorsque vous demandez l'exécution de la commande *`Format`* afin de formater un disque, vous devez spécifiez la lettre du lecteur qui doit être formaté. Il s'agit là d'un paramètre, d'une information supplémentaire nécessaire pour l'exécution de la commande. Voici la syntaxe permettant d'invoquer une fonction ou une procédure et de préciser les paramètres attendus par celle-ci :

```
NomFonction  Param1, Param2, ParamN
```

Remarquez que chacun des paramètres sont séparés les uns des autres par une virgule (,). Notez également que certaines fonctions attendent des paramètres optionnels. Ainsi, si vous spécifiez une valeur pour ces paramètres, la fonction utilisera cette valeur mais utilisera une valeur définie par défaut à l'interne si aucune valeur n'est spécifiée pour ces paramètres.

Finalement, une fonction peut renvoyer une valeur de retour. Une valeur de retour peut être le fruit d'un calcul ou d'une opération quelconque. Une valeur de retour peut également être une information indiquant si la fonction s'est exécutée correctement ou si elle a échouée. Vous retrouverez au sein du présent chapitre la signification des valeurs de retour des fonctions intrinsèques à *VBScript*. Voici la syntaxe permettant d'invoquer une fonction et d'en récupérer la valeur de retour :

```
MaVariable = NomFonction()
```

ou, dans le cas de méthodes paramétrables :

```
MaVariable = NomFonction(Param1, Param2, ParamN)
```

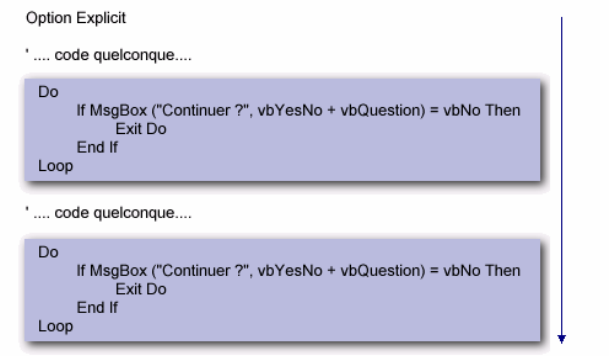
Dans laquelle syntaxe la valeur de retour de la méthode a été stockée au sein de la variable *`MaVariable`*. Notez l'utilisation des parenthèses lorsque la valeur de retour de la fonction est attendue et stockée au sein d'une variable. Votre script n'est pas obligé de récupérer les valeurs de retour des fonctions. Lorsqu'une fonction prévoit une valeur de retour et que vous ne désirez pas la stocker, utilisez un appel de procédure normal sans utiliser les parenthèses.

Création de fonctions et procédures

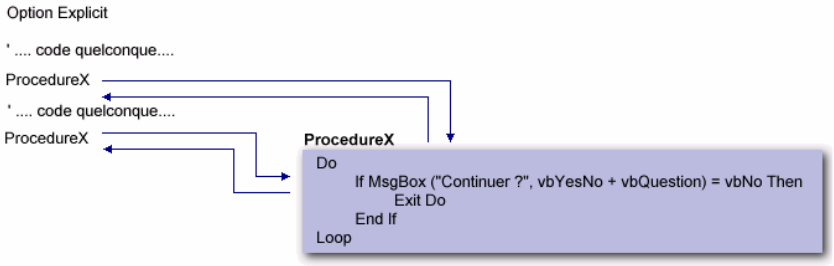
Lorsque vous écrirez vos scripts, vous rencontrerez certainement des situations dans lesquelles vous êtes forcé de répéter intégralement ou à quelques différences près un code précédemment tapé. La première fois que cette situation se présente, c'est comique… la seconde fois, c'est sympathique mais, ensuite, la situation devient rapidement lassante de par sa redondance. De plus, le fait de copier un même code à plusieurs endroits peu provoquer des ennuis plus sérieux. Si pour une raison ou une autre un code déjà copié à maints autres endroits devait être modifié, ces mêmes modifications devraient obligatoirement être appliquées aux autres duplicata de ce code. Le mot *redondance* prend alors tout son sens.

Heureusement, *VBScript* vous permet d'encapsuler tout code redondant au sein de fonctions ou de procédures afin de pouvoir aisément le réutiliser ultérieurement.

Le schéma ci-contre représente un script au sein de lequel le même segment de code a été copié et répété. Si une modification était à apporter au premier segment, cette modification devrait également s'effectuer sur le second segment.



Le schéma ci-contre représente un script au sein de lequel un segment de code a été intégré au sein d'une procédure. Si une modification était à apporter au code de cette procédure, cette modification serait automatiquement perceptible lors de chacun des appels de procédure.



Vous procéderez à l'exécution de vos procédures et fonctions personnalisées de la même manière que celle vous permettant d'exécuter les fonctions intrinsèques à *VBScript* tel que présenté précédemment.

Procédures

Les procédures contiennent du code que vous pouvez réutiliser régulièrement au sein de votre script sans avoir à le réécrire ou le copier. À la place, vous inscrivez le code redondant au sein d'une procédure et en demandez l'exécution à chaque endroit désiré au sein de votre script.

Le code d'une procédure doit préalablement être déclaré avant de pouvoir être référencé par le script. La déclaration d'une procédure s'effectue à l'aide de la syntaxe suivante :

Sub *NomProcédure* **(** *[ListeArguments]* **)**

< *Instructions*>

End Sub

Voici un exemple au sein duquel les quatre opérations arithmétiques de base sont appliquées sur un nombre quelconque contenu au sein de la variable `N`. Remarquez l'appel à la procédure `Continuer` exécutée afin de confirmer l'intention de l'utilisateur de continuer l'exécution du script.

```
Option Explicit
Dim N
N = 3

WScript.Echo "N + N = " & N + N
Continuer

WScript.Echo "N - N = " & N - N
Continuer

WScript.Echo "N * N = " & N * N
Continuer

WScript.Echo "N / N = " & N / N


'******************** Procédure Continuer()  *****************'
Sub Continuer()
  Dim Reponse
  Do
      Reponse = LCase(InputBox("Continuer? (o)ui, (n)on"))
  Loop While (Reponse <> "o" And Reponse <> "n")
  If Reponse = "n" Then WScript.Quit
End Sub
```

CH02\ContinuerProc.vbs

Il est possible de quitter prématurément une procédure en inscrivant simplement l'instruction **`Exit Sub`**.

Fonctions
Les fonctions contiennent du code que vous pouvez réutiliser régulièrement au sein de votre script sans avoir à le réécrire ou le copier de manière plus flexible qu'une procédure puisque les fonctions permettent de retourner une valeur de retour.

Le code d'une procédure doit préalablement être déclaré avant de pouvoir être référencé par le script. La déclaration d'une procédure s'effectue à l'aide de la syntaxe suivante :

```
Function NomFonction ( [ListeArguments] )

    < Instructions>
    NomFonction = valeur

End Function
```

Voici un exemple au sein duquel les quatre opérations arithmétiques de base sont appliquées sur un nombre quelconque contenu au sein de la variable `N`. Remarquez l'appel à la fonction `Continuer` exécutée afin de confirmer l'intention de l'utilisateur de continuer l'exécution du script.

```
Option Explicit
Dim N
N = 3

WScript.Echo "N + N = " & N + N
If Continuer() = False Then WScript.Quit

WScript.Echo "N - N = " & N - N
If Continuer() = False Then WScript.Quit

WScript.Echo "N * N = " & N * N
If Continuer() = False Then WScript.Quit

WScript.Echo "N / N = " & N / N


'******************** Procédure Continuer()  *****************'
Function Continuer()
  Dim Reponse
  Do
      Reponse = LCase(InputBox("Continuer? (o)ui, (n)on"))
  Loop While (Reponse <> "o" And Reponse <> "n")
  Continuer = (Reponse = "o")
End Function
```

CH02\ContinuerFunct.vbs

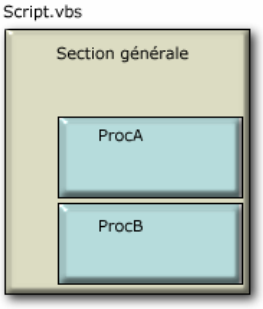
Il est possible de quitter prématurément une fonction en inscrivant simplement l'instruction **`Exit Function`**.

Bref, une fonction se distingue d'une procédure par sa capacité de retourner une valeur de retour. Notez que cette valeur de retour est précisée au sein de la fonction en assignant une valeur au nom de la fonction :

```
Continuer = (Reponse = "o")
```

Portée des variables

La portée d'une variable représente les limites au delà desquelles la variable n'est plus accessible au sein du code. Ainsi, certaines variables seront accessibles dans l'ensemble de votre script tandis que d'autres ne seront accessibles qu'à l'intérieure d'une procédure ou d'une fonction. Ce concept permet au programmeur d'éviter quantité d'erreurs imprévisibles et difficiles à diagnostiquer.



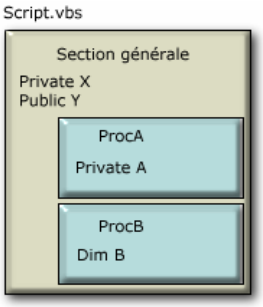
Le schéma ci-contre représente la disposition du code au sein d'un script. Le script dans son entier peut être nommé **Section générale** qui peut être sous-divisée en modules constitués de procédures ou de fonctions.

Certaines variables ne seront accessibles qu'à l'intérieur d'une procédure donnée tandis que d'autres variables seront accessibles dans l'ensemble de la section générale et des procédures la composant.

La portée d'une variable est définie par le mot-clé utilisé de la déclaration de celle-ci :

- Les mot-clés **Dim** et **Private** déclarent une variable privée qui ne sera accessible qu'à l'intérieur du module au sein duquel elle est déclarée. Ainsi, si la variable est déclarée au sein d'une procédure, elle ne sera utilisable que par le code interne à cette procédure et demeurera invisible au reste du script. Cependant, si cette variable est déclarée dans la section générale du script, elle sera visible à l'ensemble des procédures et fonctions de ce script.

- Le mot-clé **Public** déclare une variable publique qui sera accessible au module au sein duquel elle est déclarée et à l'ensemble des procédures et fonctions lui appartenant. Ainsi, si cette variable est déclarée dans la section générale du script, elle sera visible à l'ensemble des procédures et fonctions de ce script.



Le schéma ci-contre représente la déclaration de quatre variables au sein d'un script.

Puisque déclarées au niveau de la section générale, les variables `X` et `Y` sont accessibles dans l'ensemble du script.

Puisque déclarées au niveau d'une procédure ou d'une fonction, les variables `A` et `B` ne sont accessibles que dans leur procédure ou fonction respective.

Si, par exemple, la procédure `ProcA` tentait d'accéder à la variable `B` déclarée au sein de la procédure `ProcB`, un message d'erreur afficherait que la variable n'a pas été déclarée. Cependant, la procédure ProcA peut sans encombre accéder à la variable `X` déclarée au sein de la section générale du script.

De manière générale, si votre variable doit être accessible de l'ensemble des procédures et fonctions, déclarez celle-ci au sein de la section générale du script. Autrement, déclarez vos variables à l'intérieur des procédures au sein desquelles l'accès à cette variable est nécessaire.

Passage de paramètres

Les exemples précédents de procédures et fonctions ne vous permettaient pas de préciser des paramètres. Or, le passage de paramètres est une pratique fort utile dans l'élaboration de procédures et de fonctions flexibles et réutilisables.

Le passage de paramètres permet à une procédure ou à une fonction de réagir différemment selon la valeur des paramètres qu'elle a reçue. Par exemple, je peux demander à une calculatrice d'effectuer une addition sur deux valeurs numériques 7 + 4. Cependant, je peux demander à cette même calculatrice d'effectuer une même addition mais, cette fois, avec deux valeurs différentes : 6 + 5. Les instructions que doit exécuter la calculatrice demeurent les mêmes lors de l'exécution des deux opérations pourtant différentes puisque seules les valeurs changent. Dans de telles situations, le programmeur a intérêt à créer une fonction qui, comme la calculatrice, exécutera toujours les mêmes instructions en y modifiant seulement les valeurs.

Vous pouvez déclarer une fonction ou une procédure et spécifier qu'elle nécessite le passage de certains paramètres afin de s'exécuter adéquatement en précisant ces paramètres entre les parenthèses de la déclaration de la fonction ou de la procédure et en séparant les différents paramètres par des virgules :

```
Function|Sub NomProcedure (Param1 [, Param2] [, ParamN])
```

L'exemple suivant montre une fonction qui permettrait de prendre deux nombres et d'en retourner la somme :

```
Function Addition (Nbr1, Nbr2)
      Addition = Nbr1 + Nbr2
End Function
```

Cette fonction pourrait maintenant être exécutée au sein du script à l'aide de l'instruction suivante :

```
Dim Somme
Somme = Addition(2, 4)          ' Somme vaut désormais 6.
```

Utilisation de tableaux

Le code suivant procède à la déclaration de cinq variables contenant des valeurs numériques :

```
Dim Chiffre1, Chiffre2, Chiffre3, Chiffre4, Chiffre5
```

Pour pouvoir calculer la moyenne de ces variables, le code devrait ressembler au suivant :

```
Moyenne = (Chiffre1 + Chiffre2 + Chiffre3 + Chiffre4 + Chiffre5) \ 5
```

Imaginez maintenant le code nécessaire pour calculer cent valeurs numériques ou, pire encore, un nombre indéterminé de valeurs numériques. C'est à ce moment que les tableaux viennent à la rescousse.

Un tableau est simplement un ensemble de variables du même type se succédant au sein de l'espace mémoire et accessible sous la même appellation. Chacun des éléments du tableau est identifié par son index. L'index d'un élément correspond à la position ordinale de l'élément au sein du tableau. L'illustration suivante montre la mémoire occupée par un tableau de cinq éléments contenant chacun un prénom :

Tableau					
Index	0	1	2	3	4
Contenu	"Luc"	"Berthe"	"Yannick"	"Mario"	"Jeanne"

Ainsi, si on affiche la valeur de l'élément 2, nous obtiendrons "Yannick" :

```
Prenom = Tableau(2)
MsgBox Prenom     'Affiche "Yannick"
```

Le principal avantage d'un tableau demeure indéniablement la simplicité du code nécessaire pour parcourir l'ensemble de ses éléments :

```
Dim N
For N = 0 To 4
       Prenom = Tableau(N)
       MsgBox Prenom
Next
```

Les tableaux demeurent donc les meilleures solutions lorsque votre code nécessite le stockage d'un grand nombre de valeurs ou un nombre indéterminé de valeurs.

Déclaration et utilisation de tableaux

Le tableau est le mode de stockage vectoriel de base prévu par *VBScript*. Un tableau est déclaré en précisant entre parenthèses l'index du dernier élément désiré :

```
Dim MonTableau(9)  'Déclare un tableau de 10 éléments, de 0 à 9
```

Les valeurs sont ensuite assignées individuellement à chacun des éléments du tableau en précisant entre parenthèses l'index de l'élément que la valeur doit être assignée.

```
Dim MonTableau(9)

MonTableau(0) = "Luc"
MonTableau(1) = "Berthe"
Etc…
```

L'exemple suivant assigne des valeurs de 1 à 100 aux différents éléments d'un tableau :

```
Dim Nombres(99), N

For N = 0 To 99
      Nombres(N) = N + 1
Next
```

L'exemple suivant assigne les lettres de 'A' à 'Z' aux différents éléments d'un tableau :

```
Dim Lettres(25), N

For N = 0 To 25
      Lettres(N) = Chr(N + 65)   '65 = Caractère ASCII du 'A'
Next
```

L'exemple suivant invite l'utilisateur à saisir dix nombres et en calcule ensuite la moyenne :

```
Dim Nombres, N, Total

For N = 0 To 9
    Nombres(N) = CInt(InputBox("Entrez le nombre " & N + 1))
Next

Total = 0

For N = 0 To 9
    Total = Total + Nombres(N)
Next

MsgBox "La moyenne est : " & Total / 10
```

Un tableau peut également posséder un nombre indéterminé d'éléments. Ce type de tableau, nommé tableau dynamique, est utile lorsque le code ne peut connaître d'avance le nombre d'éléments qui seront nécessaires. Pour créer un tableau dynamique, il est nécessaire d'omettre l'index de l'élément supérieur à la déclaration du tableau :

```
Dim MonTableau()    'Déclare un tableau dynamique
```

Cependant, ce tableau n'est pas utilisable puisqu'il est vide et est considéré comme contenant aucun élément. Toute forme d'accès à un des éléments de ce tableau provoquerait immanquablement une erreur. Le message d'erreur *"Indice en dehors de la plage"* s'affichera alors. Il est donc nécessaire de redimensionner le tableau avant de l'utiliser en utilisant le mot-clé réservé `Redim` et en précisant les nouvelles dimensions du tableau.

```
Dim MonTableau()
Redim MonTableau(1) 'Redimensionne le tableau à 2 éléments, 0 et 1
```

Cette technique possède cependant le désavantage d'effacer le contenu de tout élément existant au sein du tableau.

```
Dim MonTableau()
Redim MonTableau(1)

MonTableau(0) = "Luc"
MonTableau(1) = "Berthe"

Redim MonTableau(9)

MsgBox MonTableau(0) 'Affiche "" puisque Redim en a effacé le
                     'contenu.
```

Si votre code nécessite cependant la conservation des valeurs existantes, le redimensionnement doit inclure le mot-clé `Preserve`. Ce dernier assure la conservation du contenu existant.

```
Dim MonTableau()
Redim MonTableau(1)

MonTableau(0) = "Luc"
MonTableau(1) = "Berthe"

Redim Preserve MonTableau(9)

MsgBox MonTableau(0)   'Affiche "Luc".
```

Maintenant que votre code peut créer des tableaux dynamiques, il pourrait lui arriver de ne pas savoir exactement combien d'éléments constituent le tableau. Cette information peut être récupérée à l'aide de la fonction `UBound` qui retourne l'index du dernier élément du tableau. L'exemple suivant parcours et affiche l'ensemble des éléments d'un tableau sans en connaître préalablement les dimensions :

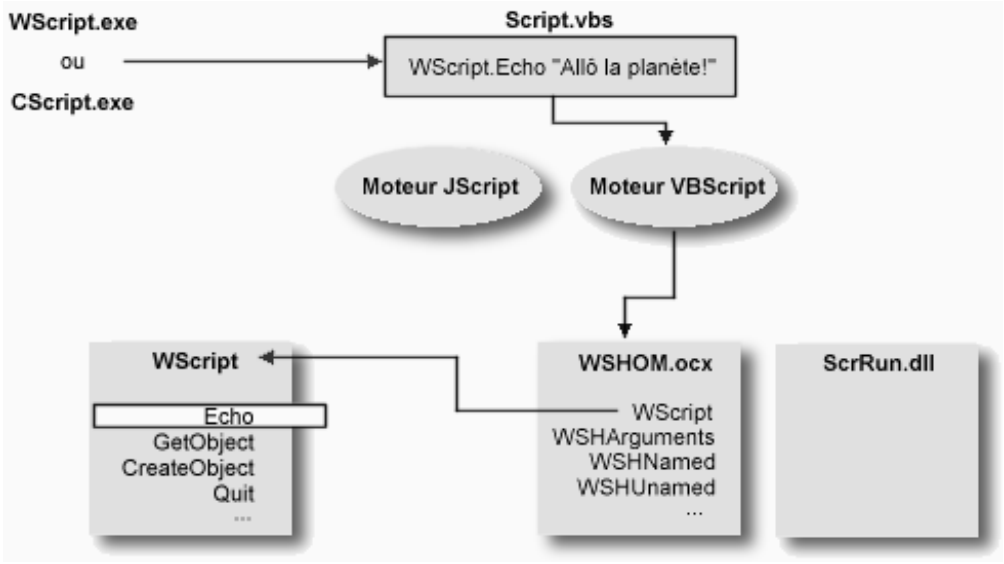
```
For N = 0 To UBound(MonTableau)
      MsgBox MonTableau(N)
Next
```

Objets, propriétés et méthodes

Les langages de script de *Windows Script Host* utilisent des objets prédéfinis afin de pouvoir exécuter un ensemble de fonctionnalités. Avant de pouvoir utiliser ces objets au sein de vos scripts, vous devez impérativement comprendre certains concepts associés à la programmation orientée-objets.

Objets

Un objet peut être perçu comme un lot de fonctionnalités regroupés thématiquement sous un même nom. L'objet contient le code nécessaire à l'exécution des fonctionnalités que vous utiliserez mais ce code ne vous est pas accessible : vous pouvez l'utiliser mais vous ne pouvez pas le voir. Autre analogie, lorsque vous utilisez la commande `format` afin de formater votre disque dur, vous utilisez une fonctionnalité de laquelle il vous est impossible d'en consulter le code.



Un ou plusieurs objets ainsi que ses fonctionnalités associées sont généralement compilés en fichiers binaires et peuvent être du type Dynamic Link Librairie *(DLL)* ou ActiveX Control *(OCX)*. Puisque les objets regroupent des fonctionnalités de manière thématique, la première étape avant d'écrire une instruction au sein d'un script est de connaître l'objet possédant les fonctionnalités que l'on désire utiliser. Les *modèles d'objets* agissent en frais de documentation permettant de connaître la liste d'objets disponibles au sein d'un fichier binaire. Lorsque vous avez localisé l'objet qui devra être utilisé, vous pouvez prendre connaissance de l'ensemble de ses fonctionnalités qui vous sont exposés sous l'appellation de *Méthodes* et de *Propriétés*.

Méthodes

Les objets exposent leurs fonctionnalités à votre script à l'aide de leurs *méthodes*. Lorsque votre script invoque une méthode d'un objet, votre script demande à cet objet d'exécuter une de ses fonctionnalités.

Voici une première syntaxe permettant d'invoquer une méthode :

```
Objet.Méthode
```

Remarquez que le point (.) est utilisé afin de séparer le nom de l'objet du nom de sa méthode à invoquer. Cet appel d'une méthode met en oeuvre la syntaxe la plus simple puisque la méthode ne nécessite pas que des paramètres d'entrée ne lui soient spécifiés. Les paramètres représentent des informations supplémentaires que la méthode pourrait attendre afin de préciser certains aspects de l'appel. Ainsi, lorsque vous demandez l'exécution de la commande *`Format`* afin de formater un disque, vous devez spécifiez la lettre du lecteur qui doit être formaté. Il s'agit là d'un paramètre, d'une information supplémentaire nécessaire pour l'exécution de la commande. Voici la syntaxe permettant d'invoquer une méthode et de préciser les paramètres attendus par celle-ci :

```
Objet.Méthode  Param1, Param2, ParamN
```

Remarquez que chacun des paramètres sont séparés les uns des autres par une virgule (,). Notez également que certaines fonctions attendent des paramètres optionnels. Ainsi, si vous spécifiez une valeur pour ces paramètres, la méthode utilisera cette valeur mais utilisera une valeur par défaut si aucune valeur n'est spécifiée pour ces paramètres.

Finalement, une méthode peut renvoyer une valeur de retour. Une valeur de retour peut être le fruit d'un calcul ou d'une opération quelconque. Une valeur de retour peut également être une information indiquant si la méthode s'est exécutée correctement ou si elle a échouée. Vous retrouverez au sein de la documentation d'un objet la signification des valeurs de retour de ses méthodes. Voici la syntaxe permettant d'invoquer une méthode et d'en récupérer la valeur de retour :

```
MaVariable = Objet.Méthode()
```

ou, dans le cas de méthodes paramétrables :

```
MaVariable = Objet.Méthode(Param1, Param2, ParamN)
```

Dans laquelle syntaxe la valeur de retour de la méthode a été stockée au sein de la variable *`MaVariable`*. Notez l'utilisation des parenthèses lorsque la valeur de retour de la méthode est attendue et stockée au sein d'une variable. Votre script n'est pas obligé de récupérer les valeurs de retour des méthodes. Si une méthode prévoit une valeur de retour et que vous ne désirez pas la stocker, utilisez un appel de méthode normal sans utilisation des parenthèses.

Propriétés

Les objets exposent leurs données et leurs caractéristiques à votre script à l'aide de leurs *propriétés*. Ainsi, pour pouvoir accéder aux données et caractéristiques d'un objet, vous devez savoir comment lire et écrire au sein d'une propriété de cet objet. Voici la syntaxe permettant de lire la valeur d'une propriété d'un objet :

```
MaVariable = Objet.Propriété
```

Par laquelle syntaxe la valeur de la Propriété *Propriété* de l'objet *Objet* a été stockée au sein de la variable *MaVariable*.

Voici la syntaxe permettant d'écrire la valeur d'une propriété d'un objet :

```
Objet.Propriété = Valeur
```

Par laquelle syntaxe la valeur *Valeur* a été stockée au sein de la propriété *Propriété* de l'objet *Objet*. La valeur peut être constituée de :

- **Un littéral**
  Vous pouvez assigner une valeur littérale à une propriété telle une valeur numérique comme le nombre 16 ou une chaîne de caractères comme "chien".

- **La propriété d'un autre objet**
  Vous pouvez assigner la valeur d'une variable ou d'une propriété d'un autre objet.

  ```
  Objet1.Propriété = Objet2.Propriété
  ```

- **Une expression**
  Vous pouvez assigner une expression comme 12 + 4 ou "chien" concaténé avec "chat". Le résultat final de cette expression sera d'abord évalué par le moteur de script avant d'être assigné à la propriété.

- **La valeur de retour renvoyée par une méthode**
  Vous pouvez assigner la valeur de retour d'une méthode. La méthode sera d'abord exécutée par le moteur de script avant d'assigner sa valeur de retour à la propriété.

  ```
  Objet.Propriété = Méthode([arg])
  ```

Voici un simple exemple de l'utilisation de méthodes et de propriétés de différents objets du modèle d'objets de *Window Script Host*. Voyez le chapitre 3 afin de consulter la documentation complète au sujet de ce modèle d'objets.

```
Option Explicit

Dim objNet

Set objNet = WScript.CreateObject("WScript.Network")
Domaine = objNet.UserDomain
Utilisateur = objNet.UserName
Ordinateur = objNet.ComputerName

WScript.Echo "Bienvenue " & Domaine & "\" & Utilisateur & "."
WScript.Echo "Vous utilisez l'ordinateur " & Ordinateur & "."
```

CH02\Utilisateur.vbs

Notez d'abord l'utilisation des parenthèses lors de l'appel de la méthode `CreateObject` puisque la valeur de retour est attendue par le script et stockée au sein de la variable `objNet`.

Notez ensuite l'assignation de différentes propriétés de l'objet `objNet` aux différentes variables `Domaine`, `Utilisateur` et `Ordinateur`.

Finalement, notez les appels de la méthode `Echo` de l'objet `WScript` qui ne gèrent pas les valeurs de retour de cette méthode. Ainsi, la syntaxe ne prévoit pas de parenthèses pour ces appels de méthodes.

Création et destruction des objets

Lorsqu'un programmeur conçoit un objet afin que vous puissiez le réutiliser au sein de vos scripts, celui-ci procède à la création de ce qu'on appelle une **classe**. Une classe est un modèle, un plan, un *blueprint* d'un objet. Une classe défini le comportement d'un objet, la liste des propriétés et des méthodes exposées par cet objet ainsi que le code qui devra être exécuté lors de l'invocation de l'une des ces méthodes ou lors de l'utilisation de l'une de ces propriétés. Par contre, avant d'être adéquatement utilisée au sein de votre script, vous devez créer une instance de cette classe en procédant à ce que l'on appelle l'**instanciation** d'un objet.

Afin de bien comprendre ces concepts plutôt abstraits, comparez-les à la construction d'une maison : l'architecte conçoit le plan d'une maison. Ce plan régi la construction éventuelle de maisons qui seront tous semblables si elles suivent à la lettre le plan de l'architecte. Cependant, le plan n'est pas habitable puisque le plan n'est pas la maison. Au même titre, la classe n'est pas l'objet.

Avant de pouvoir habiter la maison, vous devez en construire une instance selon le plan. C'est l'étape de l'instanciation d'un objet, c'est-à-dire la création d'un objet selon le plan défini par la classe. Maintenant, si vous décidez de construire plusieurs maisons identiques selon le même plan et que vous décidez d'en repeindre une en rouge, les autres maisons ne seront pas affectées et demeureront de la même couleur originale. Le même concept s'applique aux objets : si vous procédez à plusieurs instances d'une même classe et que vous modifiez les propriétés d'un de ces objets, les autres instances n'en seront aucunement affectées.

Création d'un objet

Vous devez instancier un objet avant de pouvoir en utiliser les méthodes et propriétés. Vous procédez à l'instanciation d'un objet au sein de vos scripts à l'aide de l'instruction `CreateObject`. Cette instruction est intrinsèque au langage VBScript.

```
Set Variable_Objet = CreateObject(Objet_ProgID)
```

Par exemple, le script suivant procède à l'instanciation d'un objet de type *Network* :

```
Set objNet = CreateObject("WScript.Network")
```

L'objet est désormais utilisable au sein de votre script sous l'appellation *objNet*. L'objet créé à partir de la classe *Network* possède entre autres la méthode *MapNetworkDrive* permettant de connecter un lecteur réseau :

```
Option Explicit

Dim objNet

Set objNet = CreateObject("WScript.Network")
objNet.MapNetworkDrive "Z:", "\\serveur\partage"
WScript.Echo "Le lecteur Z: est connecté!"
```

CH02\map.vbs

L'objet `WScript` est un objet intrinsèque au modèle d'objets de Windows Script Host et n'a pas besoin d'être instancié avant d'être utilisé. Une instance de cet objet est omniprésente dans tout les scripts et, à vrai dire, cet objet ne peut tout simplement pas être instancié et apporte l'exception à la règle.

Destruction d'un objet

Lorsqu'un objet est créé à l'aide de l'instruction `CreateObject`, la structure de la classe correspondante est chargée en mémoire. Lorsque l'objet n'est plus utilisé, il faut rendre au système d'exploitation l'espace mémoire consommé par la création des différents objets. Le script procède alors à la destruction de l'objet. Cet objet devient alors inutilisable et il devient alors impossible d'en utiliser ni ses méthodes ni ses propriétés. On procède à la destruction explicite d'un objet comme suit :

```
Set Variable_Objet = Nothing
```

Notez cependant que tout objet créé par un script est automatiquement détruit par *Windows Script Host* lors de la fin de l'exécution de ce script. La suppression explicite des objets à l'aide du mot-clé `Nothing` est cependant recommandée en tant que bonne habitude de programmation structurée.

La liste des objets utilisables sur votre ordinateur peut être connue en consultant les différentes sous-clés de la clé `HKEY_CLASSES_ROOT` de votre base de registres. Les objets sont listés après la liste des associations d'extensions de fichier dont les noms de clés sont précédés d'un point (.). Chacun des objets utilisables sont listés sous le format *NomLibrairie.NomObjet*.

Utilisation d'un objet existant

Il est possible de référencer un objet existant à l'aide de l'instruction `GetObject`. Aucune nouvelle instance d'une classe n'est créée mais il est alors possible d'obtenir une référence sur un objet existant.

```
Set Variable_Objet = GetObject(Fichier)
```

Par exemple, le script suivant récupère une référence sur un objet de type *Winword.Document* correspondant au fichier *« c:\document.doc »* :

```
Set objDoc = GetObject("c:\document.doc")
```

L'objet est désormais utilisable au sein de votre script sous l'appellation *objDoc*. L'exemple suivant provoque l'affichage du fichier *« c:\document.doc »* au sein de Word si ce fichier existe :

```
Set objDoc = GetObject("c:\document.doc")
objDoc.Parent.Visible = True
```

CH02\Document.vbs

Gestion des erreurs

Pour diverses raisons, votre code peut parfois générer des erreurs soulevées par le système. Une erreur peut survenir lorsque votre script tente d'accéder à un lecteur temporairement inaccessible *(un lecteur de disquettes ne possédant pas de disquette)*, lorsque votre script tente d'exécuter une instruction non-conforme *(division par zéro)*, lorsque votre script tente d'accéder à un chemin réseau introuvable, etc.

Cependant, il ne faut pas nécessairement percevoir les erreurs comme étant de méchantes bêtes surgissant toujours du noir au moment le plus inopportun. Les erreurs sont souvent des messages prévenant le programmeur d'une situation quelconque et pas forcément désastreuse. Par exemple, le script suivant s'exécute bien la première fois mais génère une erreur dès qu'il est ré-exécuté :

```
Option Explicit

Dim objNet

Set objNet = CreateObject("WScript.Network")
objNet.MapNetworkDrive "Z:", "\\serveur\partage"
MsgBox "Le lecteur Z: est connecté!", vbInformation
```

CH02\map.vbs

Ce script génère une erreur lorsqu'il est ré-exécuté tout simplement parce que le lecteur "z:" est déjà connecté à un chemin réseau et ne peut être utilisé une seconde fois. Est-ce pour autant tragique ? Il suffit au programmeur de prévoir ces situations au sein desquelles des erreurs peuvent survenir et de les traiter avec une attention particulière.

L'instruction On Error Resume Next

VBScript prévoit une instruction permettant au code de faire fît des erreurs pouvant survenir au sein du script. Si, considérant l'exemple précédant, l'instruction `MapNetworkDrive` génère une erreur puisque le lecteur a déjà été connecté, l'instruction `On Error Resume Next` précisera à *VBScript* de ne pas en tenir compte et de passer outre.

```
Option Explicit
Dim objNet

Set objNet = CreateObject("WScript.Network")

On Error Resume Next

objNet.MapNetworkDrive "Z:", "\\serveur\partage"
MsgBox "Le lecteur Z: est connecté!", vbInformation
```

L'objet Err

Maintenant, avec une telle instruction, il est désormais possible d'exécuter un script au grand complet sans qu'aucune erreur ne soit jamais générée. Youppi ! Comme il a été précisé précédemment, les erreurs ne doivent pas être perçues comme des événements tragiques mais plutôt comme des messages qu'il suffit de traiter adéquatement. Ainsi, il ne suffit pas de spécifier à l'hôte de script de passer outre les erreurs mais il demeure plus sage de vérifier si une erreur est survenue et de réagir en conséquence.

L'objet `Err` représente l'erreur produite par la dernière instruction exécutée et vous permet d'en connaître la source, le numéro, la description, etc. En testant adéquatement l'objet `Err` à l'aide d'une simple structure `If`, il est possible de savoir si l'instruction précédemment exécutée a générée une erreur ou non.

```
Option Explicit
Dim objNet

Set objNet = CreateObject("WScript.Network")

On Error Resume Next

objNet.MapNetworkDrive "Z:", "\\serveur\partage"

If Err.Number Then
      MsgBox "Le lecteur Z: était déjà connecté!", vbExclamation
Else
      MsgBox "Le lecteur Z: est connecté!", vbInformation
End If
```

Notez que l'exploit est réalisable seulement si l'instruction `On Error Resume Next` a été activé sinon l'hôte de script interrompra l'exécution du script sur la ligne de code générant une erreur.

Ainsi, si la propriété `Number` de l'objet `Err` retourne zéro (0), cela signifie qu'aucune erreur n'a été générée par la dernière instruction générée. Sinon, la propriété `Err.Number` possédera comme valeur le numéro de l'erreur correspondante. L'objet `Err` expose les membres importants suivants :

Propriété	Description
Number	Retourne le numéro de la dernière erreur survenue. Retourne zéro (0) si la dernière instruction n'a générée aucune erreur.
Source	Retourne le nom de la librairie ayant générée la dernière erreur survenue.
Description	Retourne le nom de la description associé à la dernière erreur survenue.
Clear	Procède à la suppression de toute information concernant la dernière erreur survenue.

L'instruction On Error Goto 0

L'instruction `On Error Goto 0` permet d'annuler tout traitement des erreurs. Ainsi, si une erreur survient après cette instruction, l'hôte interrompra le script et générera un message d'erreur. Ce comportement de l'hôte est souhaitable puisque nous désirons connaître les erreurs non-gérées que pourrait générer notre script afin de le rendre le plus parfait que possible.

```
Option Explicit
Dim objNet

Set objNet = CreateObject("WScript.Network")

On Error Resume Next
objNet.MapNetworkDrive "Z:", "\\serveur\partage"

If Err.Number Then
    MsgBox "Le lecteur Z: était déjà connecté à un chemin ▼
      réseau!", vbExclamation
Else
    MsgBox "Le lecteur Z: est connecté!", vbInformation
End If

On Error Goto 0

Set objNet = Nothing
```

CH02\map avec traitement des erreurs.vbs