INTRODUCTION AU

LANGAGE VBSCRIPT

Serge Tahé, janvier **2002**

Hébergé par developpez.com :

Introduction

Ce document a pour but de fournir les bases du langage vbscript ainsi que des exemples d'utilisation dans différents domaines. VBScript est un langage de script sous Windows. Il peut fonctionner dans différents conteneurs tels

- *Windows Scripting Host* pour une utilisation directe sous Windows notamment pour écrire des scripts d'administration système
- *Internet Explorer*. Il est alors utilisé au sein de pages HTML auxquelles il amène une certaine interactivité impossible à atteindre avec le seul langage HTML.
- *Internet Information Server* (IIS) le serveur Web de Microsoft sur NT/2000 et son équivalent *Personal Web Server* (PWS) sur Win9x. Dans ce cas, vbscript est utilisé pour faire de la programmation côté serveur web, technologie appelée ASP (*Active Server Pages*) par Microsoft.

Par ailleurs, VBSCRIPT étant un langage dérivé de Visual Basic pour Windows, il peut servir d'introduction à ce langage parmi les plus répandus dans le domaine Windows ainsi qu'à la version *Application* de VB, appelée **VBA** (Visual Basic pour Applications). VBA est utilisé par exemple dans tout la suite Office de Microsoft notamment dans Excel. Ainsi VBSCRIPT est une voie d'entrée au développement dans un vaste domaine d'applications windows.

VBScript n'est pas un langage à objets même s'il en a une certaine coloration. La notion d'héritage, par exemple, n'existe pas. Il peut cependant utiliser les objets mis à sa disposition par le conteneur dans lequel il s'exécute ainsi que plus généralement les composants ActiveX disponibles sur la machine Windows. C'est cet aspect qui donne sa puissance à VBScript, langage qui intrinsèquement est assez pauvre mais qui grâce aux objets mis à sa disposition peut rivaliser avec des langages de script au départ plus riches tels Perl, Javascript, Python. C'est un langage simple à apprendre, à utiliser et qui ouvre la voie à l'utilisation de Visual Basic pour Windows dont il est directement dérivé.

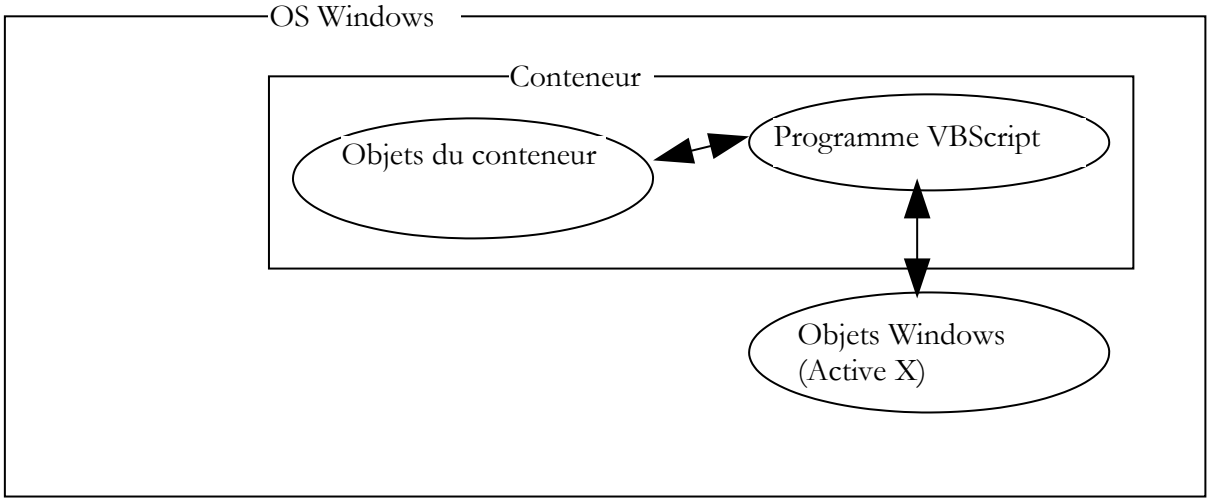
Ce document n'est pas un cours d'algorithmique. L'art de la programmation est supposé acquis. Un travail de lecture actif est nécessaire. La meilleure façon d'utiliser ce document est probablement de tester sur sa propre machine les exemples qui y sont donnés. Le conteneur WSH est normalement livré en standard avec le système Windows. La version la plus récente est disponible gratuitement sur le site de Microsoft (http://www.microsoft.com). Pour trouver l'URL exacte permettant le téléchargement de WSH, on pourra chercher les mots clés "Windows Scripting" avec un moteur de recherche sur le Web. Parmi les réponses, on devrait trouver l'URL de téléchargement de WSH.

Serge Tahé, janvier 2002

1 Les contextes d'exécution de VBSCRIPT

1.1 Introduction

Un programme *vbscript* ne s'exécute pas directement sous Windows mais dans un conteneur qui lui fournit un contexte d'exécution et un certain nombre d'objets qui lui sont propres. Par ailleurs, le programme vbscript peut utiliser des objets mis à sa disposition par le système Windows, objets appelés objets ActiveX.



Dans ce document, nous utiliserons deux conteneurs : *Windows Scripting Host* appelé couramment WSH et le navigateur *Internet Explorer* appelé parfois par la suite IE. Il en existe bien d'autres. Ainsi, les applications MS-Office sont des conteneurs pour un dérivé de VB appelé VBA (Visual Basic pour Applications). Il existe par ailleurs de nombreuses applications windows qui permettent à leurs utilisateurs de dépasser les limites de celle-ci en leur permettant de développer des programmes s'exécutant au sein de l'application et utilisant les objets propres à celle-ci.

Le conteneur dans lequel s'exécute le programme vbscript joue un rôle primordial :

- les objets mis à disposition du programme vbscript par le conteneur changent d'un conteneur à l'autre. Ainsi WSH met à disposition d'un programme vbs un objet appelé *WScript* qui donne accès, par exemple, aux partages et imprimantes réseau de la machine hôte. IE lui, met à disposition du programme vbs, un objet appelé *document* qui représente la totalité du document HTML visualisé. Le programme vbs va alors pouvoir agir sur ce document. Excel lui met à disposition d'un programme VBA des objets représentant des classeurs, des feuilles de classeurs, des graphiques, etc.... en fait tous les objets manipulés par Excel.
- si les objets d'un conteneur donnent toute sa puissance à un programme vbscript, il peut parfois en limiter certains domaines. Ainsi un programme vbscript exécuté dans le navigateur IE ne peut pas accéder au disque de la machine hôte, ceci pour des raisons de sécurité.

Donc, lorsqu'on parle de programmation vbscript, il faut indiquer dans quel conteneur le programme est exécuté.

Sous windows, vbscript n'est pas le seul langage utilisable dans les conteneurs WSH ou IE. On peut par exemple utiliser JScript (=JavaScript), PerlScript (=Perl), Python, ... Nombre de ces langages semblent de prime abord supérieurs à vbscript. Mais ce dernier a cependant de sérieux atouts :

- VB et ses déclinaisons VBSCRIPT et VBA sont très répandues sur les machines windows. Connaître ce langage paraît indispensable.
- C'est davantage les objets utilisables par un programme que le langage utilisé par celui-ci qui font sa puissance. Or nombre de ces objets sont fournis par les conteneurs et non par les langages eux-mêmes.

Un inconvénient de VBSCRIPT est qu'il n'est pas portable sur un système autre que Windows, par exemple Unix. Ses concurrents Javascript, Perl, Python eux le sont. Si on doit travailler sur des systèmes hétérogènes, il peut être intéressant voire indispensable d'utiliser le même langage sur les différents systèmes.

1.2 Le conteneur WSH

Le conteneur WSH (Windows Scripting Host) permet l'exécution, au sein de Windows, de programmes écrits en divers langages : vbscript, javascript, perlscript, python, ... Il existe une norme à respecter pour qu'un langage puisse être utilisé au sein de WSH. Tout langage respectant cette norme est candidat à l'exécution au sein de WSH. On peut imaginer que la liste précédente des langages s'exécutant dans WSH puisse s'allonger. Un conteneur met à la disposition des programmes qu'il exécute des objets qui leur donnent leur véritable puissance. Ceci tend à gommer les différences entre langages puiqu'ils utilisent alors tous le même ensemble d'objets. Utiliser un langage plutôt qu'un autre peut devenir alors une simple affaire de goût plutôt que de performances.

L'exécution d'un programme dans WSH se fait à l'aide de deux exécutables : *wscript.exe* et *cscript.exe*. *wscript.exe* se trouve normalement dans le répertoire d'installation de windows appelé généralement *%windir%* :

```
C:\ >echo %windir%
C:\WINDOWS

C:\>dir c:\windows\wscript.exe
WSCRIPT  EXE       123 280  19/09/01  11:54 wscript.exe
```

L'exécutable *cscript.exe* se trouve lui sous *%windir%\command* :

```
C:\>dir c:\windows\cscript.* /s

Répertoire de C:\WINDOWS\COMMAND
CSCRIPT  EXE       102 450  26/06/01  17:49 cscript.exe
```

Le w de *wscript* veut dire *windows* et le c de *cscript* veut dire *console*. Un script peut être exécuté indifféremment par wscript ou cscript. La différence réside dans le mode d'affichage de messages à l'écran :

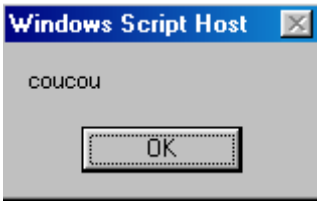
- wscript les affiche dans une fenêtre
- cscript les affiche à l'écran

Voici un script *coucou.vbs* qui affiche *coucou* à l'écran :

```
1 ' affiche coucou
2 wscript.echo "coucou"
```

Ouvrons une fenêtre DOS et exécutons-le successivement avec *wscript* et *cscript* :

DOS>`wscript coucou.vbs`



DOS>`cscript coucou.vbs`

```
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. Tous droits réservés.

coucou
```

On voit ci-dessus clairement la différence entre les deux modes. Dans ce document, nous utiliserons quasi exclusivement le mode console *cscript*. C'est le mode qui convient aux applications dites "batch" c'est à dire des applications sans interaction avec un utilisateur au clavier. On notera deux points dans les résultats précédents :

1. On a supposé que les exécutables wscript.exe et cscript.exe étaient tous les deux dans le "PATH" de la machine, ce qui permet de les lancer en tapant simplement leurs noms. Si ce n'était pas le cas, il aurait fallu écrire ici :
   ```
   DOS>c:\windows\wscript coucou.vbs
   DOS>c:\windows\command\cscript coucou.vbs
   ```
2. On notera que la version de wsh utilisée dans cet exemple et dans la suite du document est la version 5.6.
3. Le fichier source du script a le suffixe *.vbs*. C'est le suffixe désignant un script *vbscript*, un script *javascript* étant lui désigné par le suffixe *.js*.

Le programme *cscript* a diverses options de lancement qu'on obtient en lançant *cscript* sans arguments :

```
DOS>cscript

Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. Tous droits réservés.

Utilisation : CScript scriptname.extension [option...] [arguments...]

Options:
 //B         batch : Supprime l'affichage des invites et des erreurs de scripts
 //D         Activer le débogage
 //E:engine  Utiliser le moteur pour l'exécution de script
 //H:CScript Remplace l'environnement d'exécution de scripts par défaut par CScript.exe
  //H:WScript  tRemplace  l'environnement  d'exécution  de  scripts  par  défaut  par  WScript.exe
(default)
 //I         Mode interactif (par défaut, contraire de l'option  //B)
 //Job :xxxx    Exécuter une tâche WSF
 //Logo      Afficher un logo (default)
  //Nologo     Empêcher l'affichage d'un logo : Aucune bannière ne s'affiche pendant la durée
d'exécution
 //S         Enregistrer les options de ligne de commande actuelles pour cet utilisateur
```

```
 //T:nn       Durée d'exécution en secondes :  Temps maximal autorisé pour l'exécution d'un
script
 //X         Exécuter un script dans le débogueur
```

L'argument *//nologo* supprime l'affichage de la bannière de wsh :

```
C:\>cscript //nologo coucou.vbs
coucou
```

1.3 La forme d'un script WSH

Nous venons de voir un premier script : *coucou.vbs*

```
1 ' affiche coucou
2 wscript.echo "coucou"
```

Nous avons indiqué que le suffixe *.vbs* du fichier désignait un script *vbscript*. Ce n'est pas une obligation. Nous aurions pu mettre le script dans un fichier de suffixe *.wsf* sous la forme suivante plus complexe :

```
1 <job id="coucou">
2   <script language="vbscript">
3     ' affiche coucou
4     wscript.echo "coucou"
5   </script>
6 </job>
```

L'exécution de ce script donne la chose suivante :

```
C:\>cscript //nologo coucou2.wsf
coucou
```

Un script WSH peut mélanger les langages :

```
 1 <job id="coucou">
 2
 3   <script language="vbscript">
 4     ' affiche coucou
 5     wscript.echo "coucou (vbscript)"
 6   </script>
 7
 8   <script language="javascript">
 9     // affiche coucou
10     WScript.echo("coucou (javascript)");
11   </script>
12
13   <script language="perlscript">
14     # affiche coucou
15     $WScript->echo("coucou (perlscript)");
16   </script>
17
18 </job>
```

L'exécution de ce script donne la chose suivante :

```
C:\>cscript //nologo coucou3.wsf
coucou (vbscript)
coucou (javascript)
coucou (perlscript)
```

Nous retiendrons les points suivants :

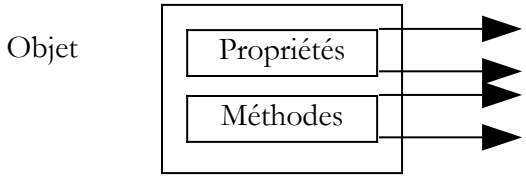
1. Le conteneur WSH n'est pas lié à un langage. Un script wsh peut mélanger les langages dans un fichier de suffixe .wsf

2. Le script est alors encadré par des balises <job id="..."> ... </job>
3. A l'intérieur de l'application (=job), les langages utilisés par les différentes portions de code sont balisées par <script language="...."> .... </script>
4. Ce langage de balisages porte un nom : XML pour e**X**tended **M**arkup **L**anguage. XML ne définit aucune balise mais des règles d'agencement de balises. Ici on devrait donc dire que le langage de balisages utilisé dans un script wsh suit la norme XML.

Par la suite, nous utiliserons exclusivement vbscript dans des fichiers .vbs.

1.4 L'objet WSCRIPT

Le conteneur WSH met à la disposition des scripts qu'il exécute un objet appelé *wscript*. Un objet a des **propriétés** et des **méthodes** :



Un objet *Obj* a des propriétés *Pi* qui fixent son état. Ainsi un objet *thermomètre* peut avoir une propriété *température*. Cette propriété est un des aspects de l'état du thermomètre. Une autre pourrait être la température maximale *Tmax* qu'il peut supporter.

L'objet *Obj* peut avoir des méthodes *Mj* qui permettent à des agents extérieurs soit de :
- connaître son état
- changer son état

Ainsi notre thermomètre, s'il est électronique, pourrait avoir une méthode *allumer* qui le mettrait en marche, une autre *éteindre* qui l'éteindrait, une autre *afficher* qui afficherait la température d'équilibre une fois celle-ci atteinte. En termes de programmation, une méthode est une fonction qui peut admettre des arguments et rendre des résultats.

En Vbscript, les propriétés Pi d'un objet Obj sont notées *Obj.Pi* et les méthodes Mj sont notées *Obj.Mj*. L'objet wscript de WSH est un objet important pour les méthodes qu'il met à disposition des scripts. Ainsi sa méthode *echo* permet d'afficher un message. La syntaxe de cette méthode est la suivante :

```
wscript.echo arg1, arg2, ..., argn
```

Les valeurs des arguments *argi* sont alors affichées dans une fenêtre (exécution par wscript) ou à l'écran (exécution par cscript sous DOS).

1.5 Le conteneur Internet Explorer

Nous avons écrit plus haut que Internet Explorer pouvait être un conteneur pour un script vbscript. Montrons-le sur un exemple simple. Suit une page HTML (**H**yper**T**ext **M**arkup **L**anguage) appelée *coucou2.htm* ne contenant pas de script vbscript.

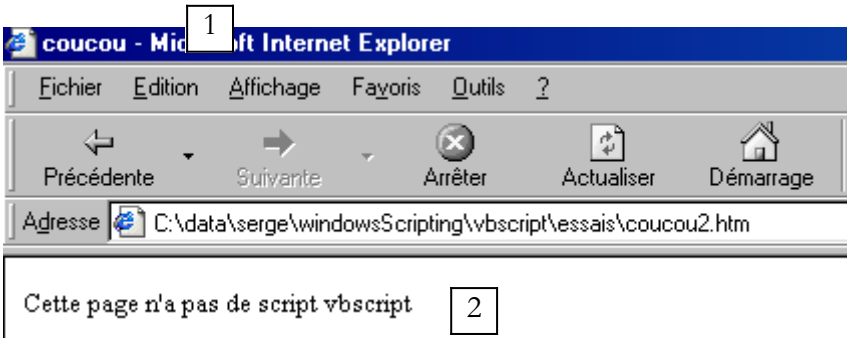
```
1 <html>
2   <head>
3     <title>coucou</title>
4   </head>
5   <body>
6     Cette page n'a pas de script vbscript
7   </body>
8 </html>
9
```

1

2

T

Son chargement direct par Internet Explorer (*Fichier/Ouvrir*) donne les résultats suivants :



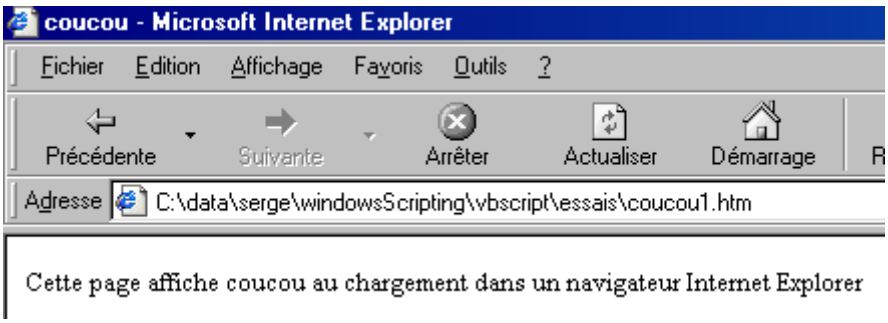
Le contenu du fichier *coucou2.htm* nous montre que HTML est un langage de balisage. Connaître le langage HTML c'est connaître ces balises. Celles-ci ont pour but principal d'indiquer au navigateur comment afficher un document. HTML ne suit pas exactement la norme XML mais en est proche.
Dans *coucou2.htm*, il y a deux informations à représenter notées 1 et 2. Nous les avons représentées également dans l'affichage qui en a été fait. C'est la balise <*title*>...</*title*> qui a fait que l'information 1 a été placée dans la barre de titre du navigateur et la balise <*body*>..</*body*> qui a fait que l'information 2 a été placée dans la partie document du navigateur.

Nous n'entrerons pas davantage dans l'étude du langage HTML. Modifions le fichier *coucou2.htm* en y introduisant un script vbscript et appelons-le maintenant *coucou1.htm* :

```
1 <html>
2   <head>
3     <title>coucou</title>
4     <script language="vbscript">
5       ' affiche coucou
6       window.alert "coucou"
7     </script>
8   </head>
9   <body>
10    Cette page affiche coucou au chargement dans un navigateur Internet Explorer
11  </body>
12 </html>
13
```

Le script vbscript a été placé dans la balise <*head*>...</*head*>. Il aurait pu être placé ailleurs. Il affiche "coucou" au chargement initial de la page. Ici, le navigateur doit être Internet Explorer car seul ce navigateur est par défaut un conteneur pour des scripts vbscript. L'affichage obtenu est alors le suivant :

suivi de l'affichage normal de la page :



Le script exécuté était le suivant :

```
<script language="vbscript">
   ' affiche coucou
  window.alert "coucou"
</script>
```

Alors que le conteneur WSH mettait à disposition du script un objet appelé *wscript* permettant de faire des affichages avec sa méthode *echo*, ici IE met à disposition du script un objet *window* permettant de faire des affichages avec la méthode *alert*. Ainsi pour afficher "coucou", on écrit *wscript.echo "coucou"* dans WSH et *window.alert "coucou"* dans IE. On peut montrer ici aussi qu'en fait on peut utiliser plusieurs langages dans le conteneur IE. Nous reprenons l'exemple déjà présenté dans WSH au sein d'une page *coucou3.htm* :

```
<html>
  <head>
    <title>coucou</title>

    <script language="vbscript">
      ' affiche coucou
      window.alert "coucou (vbscript)"
    </script>

    <script language="javascript">
      // affiche coucou
      Window.alert("coucou (javascript)");
    </script>

    <script language="perlscript">
      # affiche coucou
      $window->alert("coucou (perlscript)");
    </script>
  </head>
  <body>
    Cette page affiche coucou trois fois au chargement dans un navigateur Internet Explorer
  </body>
</html>

```

Le chargement de cette page par IE affiche tout d'abord trois fenêtres d'information :

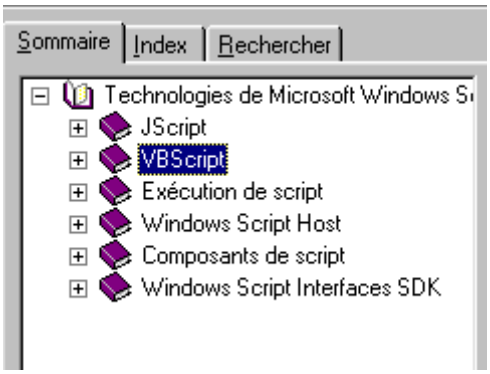
avant d'afficher la page finale :

Cette page affiche coucou trois fois au chargement dans un navigateur Internet Explorer

1.6 L'aide de WSH

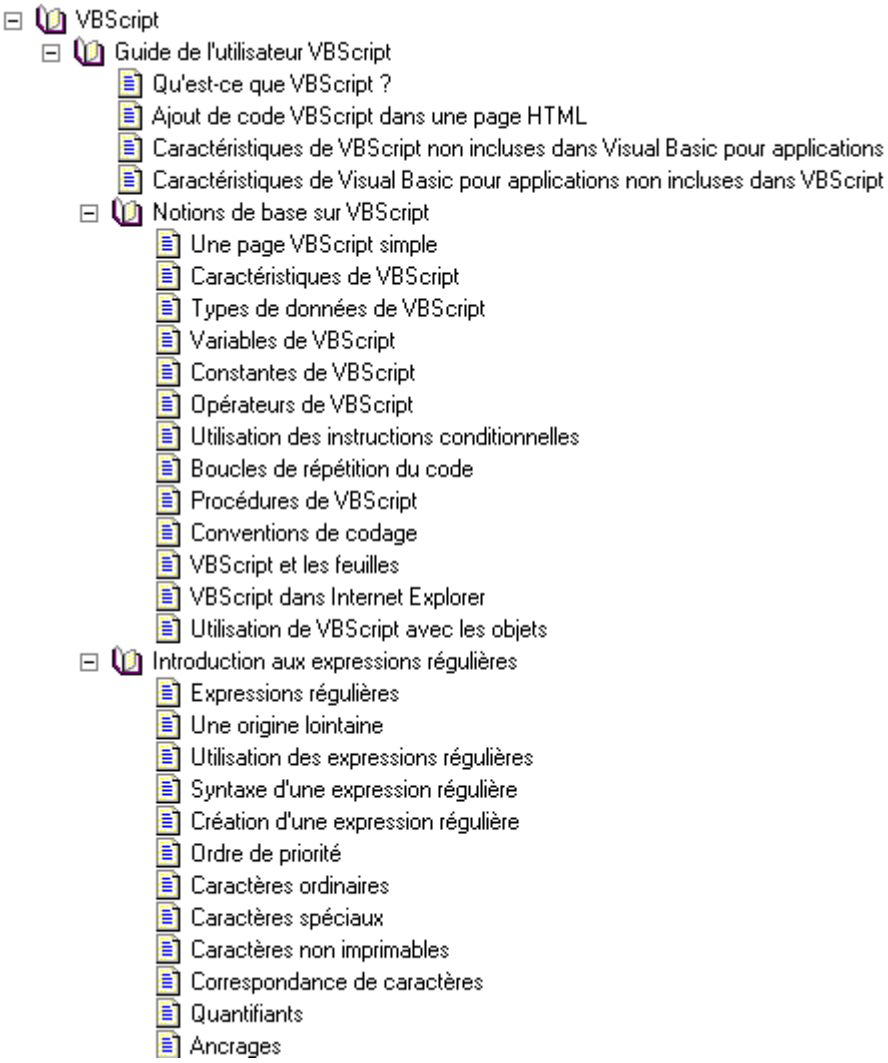
WSH vient avec un système d'aide situé habituellement dans le dossier "C:\Program Files\ Microsoft Windows Script\ScriptDocs". pour la version 5.6 de WSH, le fichier d'aide s'appelle "SCRIPT56.CHM". Il suffit de double-cliquer sur ce fichier pour avoir accès à l'aide. Il peut être pratique d'en avoir un raccourci sur son bureau.

Une fois lancé, on a quelque chose comme suit :



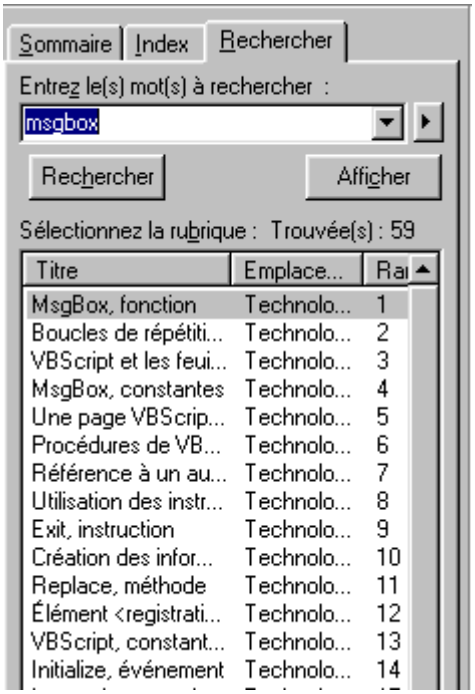
On y trouve l'aide du conteneur WSH mais également celle pour les langages vbscript et javascript. C'est un outil indispensable à la fois pour le débutant et le programmeur confirmé. Il y a plusieurs façons de travailler avec cette aide :

- on ne sait pas trop ce qu'on cherche. On veut simplement découvrir ce qui est proposé. L'onglet *Sommaire* ci-dessus peut être alors utilisé. On peut par exemple regarder ce qui est proposé pour vbscript :

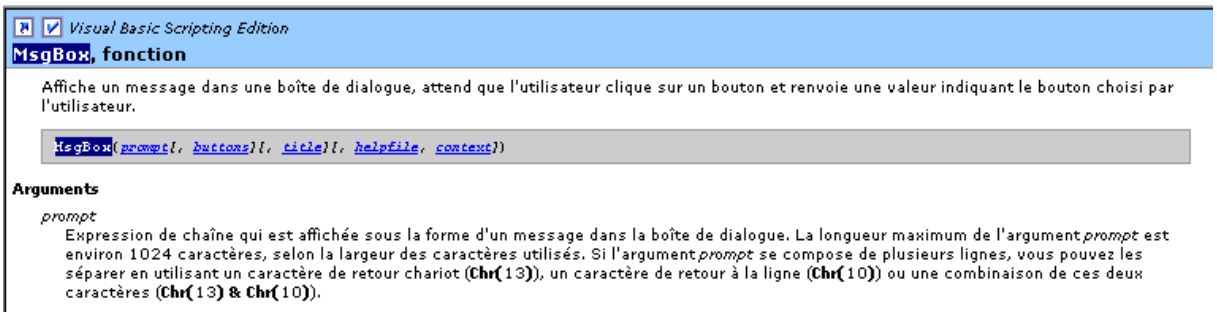


Vous découvrirez dans l'aide de VBscript de nombreuses informations qui ne sont pas dans ce document.

- vous pouvez chercher quelque chose de précis, par exemple la façon d'utiliser la fonction *msgbox* de VBscript. Utilisez alors l'onglet *Rechercher* :



L'aide ramène toutes les rubriques qui ont un rapport avec le mot recherché. En général, les premières rubriques proposées sont les plus pertinentes. C'est le cas ici où la première rubrique proposée est la bonne. Il suffit de double-cliquer dessus pour avoir l'information de cette rubrique :



2 Les bases de la programmation VBSCRIPT

Après avoir décrit les contextes d'exécution possibles pour un script vbscript, nous abordons maintenant le langage lui-même. Dans toute la suite, nous nous plaçons dans les conditions suivantes :

1. le conteneur du script est WSH
2. le script est placé dans un fichier de suffixe .vbs

Pour présenter un concept, nous opérons en général de la façon suivante :

- on introduit le concept si besoin est
- on présente un programme d'illustration avec ses résultats
- on commente les résultats et le programme si besoin est

Les conteneurs vbscript ne sont pas sensibles à la "casse" utilisée (majuscules/minuscules) dans le texte du script. Aussi pourra-t-on écrire indifféremment *wscript.echo "coucou"* ou *WSCRIPT.ECHO "coucou"*.

Les programmes présentés dans la suite font beaucoup d'écritures à l'écran aussi allons-nous présenter de nouveau les méthodes d'écriture de l'objet wscript.

2.1 Afficher des informations

Nous avons déjà utilisé la méthode *echo* de l'objet *wscript* mais ce dernier a d'autres méthodes permettant d'écrire à l'écran comme le montre le script suivant :

Programme	Résultats

```
1 ' affichages
2   wscript.echo "un"
3   wscript.stdout.write "deux"
4   wscript.stdout.write "trois" & chr(13) & chr(10)
5   wscript.stdout.write "quatre" & vbCRLF
6   wscript.stdout.writeLine "cinq"
7
```

```
un
deuxtrois
quatre
cinq
```

On notera les points suivants :

- Toute texte placé après une apostrophe est considéré comme un commentaire du script et n'est pas interprété par WSH (ligne 1).
- la méthode *echo* écrit ses arguments et passe à la ligne suivante de même que la méthode writeLine (lignes 2 et 6)
- la méthode *write* écrit ses arguments et ne passe pas à la ligne suivante (ligne 3)
- une marque de fin de ligne est représentée par la suite de deux caractères de codes ASCII 13 et 10. Ainsi ligne 4 est-elle représentée par l'expression chr(13) & chr(10) où chr(i) est le caractère de code ASCII i et & l'opérateur de concaténation de chaîne. Ainsi "chat" & "eau" est la chaîne "chateau".
- la marque de fin de ligne peut être représentée plus facilement par la constante vbCRLF (ligne 5)

2.2 écriture des instructions dans un script Vbscript

Par défaut, on écrit une instruction par ligne. Néanmoins, on peut écrire plusieurs instructions par ligne en les séparant par le caractère : comme dans *inst1:inst2:inst3*. Si une ligne est trop longue, on peut la découper en morceaux. Il faut alors que les différentes parties de l'instruction

soient terminées par les deux caractères (espace)_. Nous reprenons l'exemple précédent en réécrivant différemment les instructions :

Programme	Résultats

```
' affichages
wscript.echo "un" : wscript.stdout.write "deux"
wscript.stdout.write "trois" _
  & chr(13) & chr(10)
wscript.stdout.write "quatre" & vbCRLF : wscript.stdout.writeLine "cinq"

```

```
un
deuxtrois
quatre
cinq
```

2.3 Écrire avec la fonction msgBox

Si dans ce document, nous utilisons quasi exclusivement l'objet *wscript* pour écrire à l'écran, il existe une fonction plus sophistiquée pour afficher des informations dans une fenêtre cette fois-ci. C'est la fonction *msgbox* qui s'utilise en général avec trois paramètres :

msgbox message, icônes+boutons, titre

- message est le texte du message à afficher
- icônes+boutons (facultatif) est en fait un nombre qui indique le type d'icône et les boutons à placer dans la fenêtre du message. Ce nombre est le plus souvent la somme de deux nombres : le premier détermine l'icône, le second les boutons
- titre est le texte à placer dans la barre de titre de la fenêtre de message

Les valeurs à utiliser pour préciser l'icône et les boutons de la fenêtre d'affichage sont les suivantes :

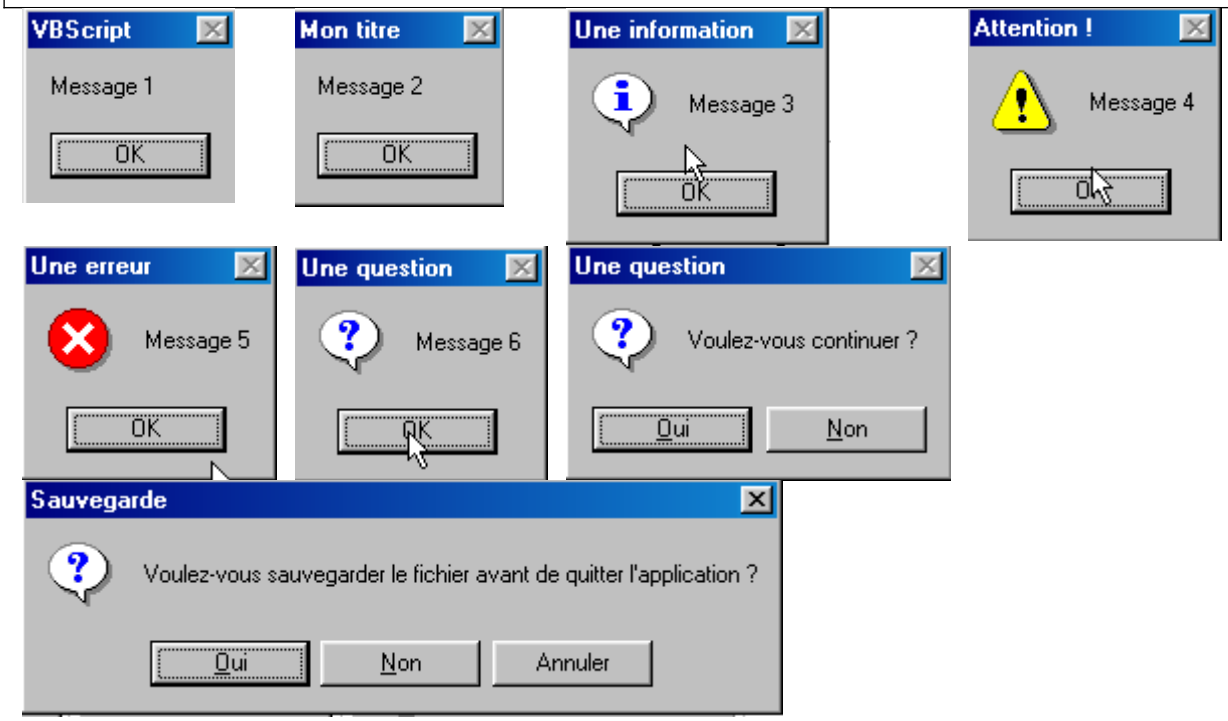
Constante	Valeur	Description
vbOKOnly	0	Affiche uniquement le bouton **OK**.
vbOKCancel	1	Affiche les boutons **OK** et **Annuler**.
vbAbortRetryIgnore	2	Affiche les boutons **Abandon**, **Réessayer** et **Ignorer**.
vbYesNoCancel	3	Affiche les boutons **Oui**, **Non** et **Annuler**.
vbYesNo	4	Affiche les boutons **Oui** et **Non**.
vbRetryCancel	5	Affiche les boutons **Réessayer** et **Annuler**.
vbCritical	16	Affiche l'icône Message critique.
vbQuestion	32	Affiche l'icône Demande d'avertissement.
vbExclamation	48	Affiche l'icône Message d'avertissement.
vbInformation	64	Affiche l'icône Message d'information.
vbDefaultButton1	0	Le premier bouton est le bouton par défaut.
vbDefaultButton2	256	Le deuxième bouton est le bouton par défaut.
vbDefaultButton3	512	Le troisième bouton est le bouton par défaut.
vbDefaultButton4	768	Le quatrième bouton est le bouton par défaut.
vbApplicationModal	0	Application modale ; l'utilisateur doit répondre au message avant de continuer à travailler dans l'application courante.
vbSystemModal	4096	Système modal ; toutes les applications sont suspendues jusqu'à ce que l'utilisateur réponde au message.

Voici des exemples :

Programme

```
1 ' msgbox
2
3 msgbox "Message 1"
4 msgbox "Message 2",,"Mon titre"
5 msgbox "Message 3",vbInformation,"Une information"
6 msgbox "Message 4",vbExclamation,"Attention !"
7 msgbox "Message 5",vbcritical,"Une erreur"
8 msgbox "Message 6",vbQuestion,"Une question"
9 msgbox "Voulez-vous continuer ?", _
10   vbQuestion+vbYesNo, "Une question"
11 msgbox "Voulez-vous sauvegarder le fichier avant de quitter l'application ?", _
12   vbQuestion+vbYesNoCancel,"Sauvegarde"
13
```

Résultats



Dans certains cas, on présente une fenêtre d'information qui est également une fenêtre de saisie. Si on pose une question, on veut par exemple savoir si l'utilisateur a cliqué sur le bouton *oui* ou sur le bouton *non*. La fonction *msgBox* rend un résultat que dans le programme précédent nous n'avons pas utilisé. Ce résultat est un nombre entier représentant le bouton utilisé par l'utilisateur pour fermer la fenêtre d'affichage :

Constante	Valeur	Bouton choisi
vbOK	1	OK
vbCancel	2	Annuler
vbAbort	3	Abandon
vbRetry	4	Réessayer
vbIgnore	5	Ignorer
vbYes	6	Oui
vbNo	7	Non

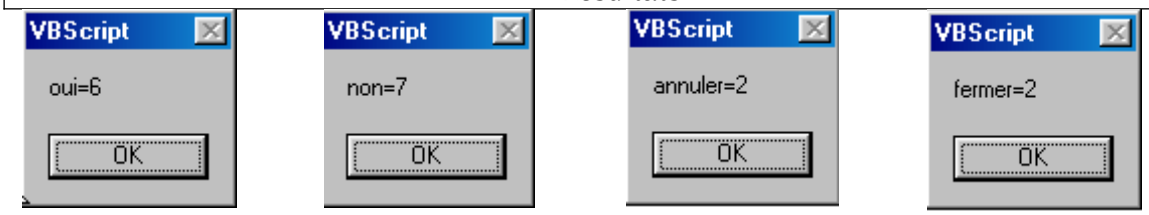
Le programme suivant montre l'utilisation du résultat de la fonction msgBox. On présente 4 fois une fenêtre avec les boutons oui, non, annuler. On répond de la façon suivante :

1. on clique sur oui
2. on clique sur non
3. on clique sur annuler
4. on ferme la fenêtre sans utiliser de bouton. Le programme montre que cela revient à utiliser le bouton Annuler.

Programme

```
1 ' msgbox
2
3 reponse=msgbox("Voulez-vous continuer ?",vbQuestion+vbYesNoCancel, "Une question")
4 msgbox "oui=" & reponse
5 reponse=msgbox("Voulez-vous continuer ?",vbQuestion+vbYesNoCancel, "Une question")
6 msgbox "non=" & reponse
7 reponse=msgbox("Voulez-vous continuer ?",vbQuestion+vbYesNoCancel, "Une question")
8 msgbox "annuler=" & reponse
9 reponse=msgbox("Voulez-vous continuer ?",vbQuestion+vbYesNoCancel, "Une question")
10 msgbox "fermer=" & reponse
```

Résultats



2.4 Les données utilisables en Vbscript

VBscript ne connaît qu'un type de données : le **variant**. La valeur d'un variant peut être un nombre (4, 10.2), une chaîne de caractères ("bonjour"), un booléen (true/false), une date (#01/01/2002#), l'adresse d'un objet, un ensemble de toutes ces données placées dans une structure appelée tableau.

Examinons le programme suivant et ses résultats :

Programme	Résultats

```
' variables
' on n'est pas obligé de déclarer les variables que le script utilis

' entiers
i=4
wscript.echo i
wscript.echo "i=" & i

' des réels
r1=10.2
wscript.echo r1
wscript.echo "r1=" & r1
r2=1.4e-2
wscript.echo r2
wscript.echo "r2=" & r2

' chaînes de caractères
c1="bonjour"
wscript.echo "c1=" & c1

' une date 10 janv 2002
d1=#10/01/2002#
wscript.echo d1
wscript.echo "d1=" & d1
' c'était en fait le 1er oct 2002, la forme des dates vbscript étant
d2=#01/10/2002#
wscript.echo d2
wscript.echo "d2=" & d2

' booléens
b1=true
wscript.echo b1
wscript.echo "b1=" & b1
b2=false
wscript.echo b2
wscript.echo "b2=" & b2

' une variable v peut changer de type de valeur au cours du temps
v=i
wscript.echo "v=" & v
v=r1
wscript.echo "v=" & v
v=c1
wscript.echo "v=" & v
v=d1
wscript.echo "v=" & v
v=b1
wscript.echo "v=" & v

' fin
wscript.quit 0
```

```
4
i=4
10,2
r1=10,2
0,014
r2=0,014
c1=bonjour
01/10/02
d1=01/10/02
10/01/02
d2=10/01/02
-1
b1=Vrai
0
b2=Faux
v=4
v=10,2
v=bonjour
v=01/10/02
v=Vrai
```

Commentaires :

- un certain nombre de langages de programmation (C, C++, Pascal, Java, C#, ...) exigent la déclaration préalable d'une variable avant son utilisation. Cette déclaration consiste à indiquer le nom de la variable et le type de données elle peut contenir (entier, réel, chaîne, date, booléen, ...). La déclaration des variables permet différentes choses :
  - connaître la place mémoire nécessaire à la variable si différents types de données nécessitent différents espaces mémoire
  - de vérifier la cohérence du programme. Ainsi si *i* est un entier et *c* une chaîne de caractères, multiplier *i* par *c* n'a aucun sens. Si le type des variables *i* et *c* a été déclaré par le programmeur, le programme chargé d'analyser le programme avant son exécution peut signaler une telle incohérence.

  Comme la plupart des langages de script à type de données unique (Perl, Python, Javascript, ...) Vbscript autorise de ne pas déclarer les variables. C'est ce qui a été fait dans l'exemple ci-dessus.

- notons la syntaxe de différentes données
  - 10.2 en ligne 10 (point décimal et non virgule). On notera qu'à l'affichage c'est 10,2 qui est affiché.

- 1.4e-2 en ligne 13 (notation scientifique). A l'affichage, c'est le nombre 0,014 qui a été affiché
- #01/10/2002# (ligne 26) pour représenter la date du 10 janvier 2002. C'est donc le format #mm/jj/aaaa# que vbscript utilise pour représenter la date *jj* du mois *mm* de l'année *aaaa*
- les booléens true et false (vrai/faux) en lignes 31 et 34. Ces deux valeurs sont représentées respectivement par les entiers -1 et 0 comme le montre l'affichage des lignes 32 et 35. Lorsqu'un booléen est concaténé à une chaîne de caractères, ces valeurs deviennent respectivement les chaînes "Vrai" et "Faux" comme le montrent les lignes 33 et 36. On remarquera au passage que l'opérateur & de concaténation peut servir à concaténer autre chose que des chaînes.

- une variable v n'ayant pas de type assigné, elle peut accueillir successivement dans le temps des valeurs de différents types.

2.5 Les sous-types du type variant

Voici que dit la documentation officielle sur les différents types de données que peut contenir un variant :

*Au-delà de la simple distinction nombre/chaîne, un **Variant** peut distinguer différents types d'information numérique. Par exemple, certaines informations numériques représentent une date ou une heure. Lorsque ces informations sont utilisées avec d'autres données de date ou d'heure, le résultat est toujours exprimé sous la forme d'une date ou d'une heure. Vous disposez aussi d'autres types d'information numérique, des valeurs booléennes jusqu'aux grands nombres à virgule flottante. Ces différentes catégories d'information qui peuvent être contenues dans un **Variant** sont des sous-types. Dans la plupart des cas, vous placez simplement vos données dans un **Variant** et celui-ci se comporte de la façon la plus appropriée en fonction de ces données.*

*Le tableau suivant présente différents sous-types susceptibles d'être contenus dans un **Variant**.*

Sous-type	Description
Empty	Le **Variant** n'est pas initialisé. Sa valeur est égale à zéro pour les variables numériques et à une chaîne de longueur nulle ("") pour les variables chaîne.
Null	Le **Variant** contient intentionnellement des données incorrectes.
Boolean	Contient True (vrai) ou False (faux).
Byte	Contient un entier de 0 à 255.
Integer	Contient un entier de -32 768 à 32 767.
Currency	-922 337 203 685 477,5808 à 922 337 203 685 477,5807.
Long	Contient un entier de -2 147 483 648 à 2 147 483 647.
Single	Contient un nombre à virgule flottante en précision simple de -3,402823E38 à -1,401298E-45 pour les valeurs négatives ; de 1,401298E-45 à 3,402823E38 pour les valeurs positives.
Double	Contient un nombre à virgule flottante en précision double de -1,79769313486232E308 à -4,94065645841247E-324 pour les valeurs négatives ; de 4,94065645841247E-324 à 1,79769313486232E308 pour les valeurs positives.
Date (Time)	Contient un nombre qui représente une date entre le 1er janvier 100 et le 31

	décembre 9999.
String	Contient une chaîne de longueur variable limitée à environ 2 milliards de caractères.
Object	Contient un objet.
Error	Contient un numéro d'erreur.

2.6 Connaître le type exact de la donnée contenue dans un variant

Une variable de type variant peut contenir des données de divers types. Il nous faut quelquefois connaître la nature exacte de ces données. Si dans un programme nous écrivons *produit=nombre1*nombre2*, nous supposons que *nombre1* et *nombre2* sont deux données numériques. Parfois nous n'en sommes pas sûrs car ces valeurs peuvent provenir d'une saisie au clavier, d'un fichier, d'une source extérieure quelconque. Il nous faut alors vérifier la nature des données placées dans *nombre1* et *nombre2*. La fonction *typename(var)* nous permet de connaître le type de données contenue dans la variable *var*. Voici des exemples :

Programme

```
1 ' types
2 ' par la suite, var est un variant qui prend différentes v
3
4 ' entier
5   var=1
6   wscript.echo "var=" & var & ",type=" & typename(var)
7
8 ' chaîne de caractères
9   var="deux"
10  wscript.echo "var=" & var & ",type=" & typename(var)
11
12 ' booléen
13  var=true
14  wscript.echo "var=" & var & ",type=" & typename(var)
15
16 ' réel
17  var=4.5
18  wscript.echo "var=" & var & ",type=" & typename(var)
19
20  ' date
21  var=#10/11/2001#
22  wscript.echo "var=" & var & ",type=" & typename(var)
23
```

Résultats

```
var=1,type=Integer
var=deux,type=String
var=Vrai,type=Boolean
var=4,5,type=Double
var=11/10/01,type=Date
```

Une autre fonction possible est *vartype(var)* qui rend un nombre représentant le type de la donnée contenue par la variable *var* :

Constante	Valeur	Description
vbEmpty	0	Empty (non initialisée)
vbNull	1	Null (aucune donnée valide)
vbInteger	2	Entier
vbLong	3	Entier long
vbSingle	4	Nombre en virgule flottante en simple précision
vbDouble	5	Nombre en virgule flottante en double précision
vbCurrency	6	Monétaire
vbDate	7	Date
vbString	8	Chaîne
vbObject	9	Objet Automation

vbError	10	Erreur
vbBoolean	11	Booléen
vbVariant	12	Variant (utilisé seulement avec des tableaux de Variants)
vbDataObject	13	Objet non Automation
vbByte	17	Octet
vbArray	8192	Tableau

Remarque Ces constantes sont spécifiées par VBScript. En conséquence, les noms peuvent être utilisés n'importe où dans votre code à la place des valeurs réelles.

Les informations ci-dessus proviennent de la documentation de VBscript. Celle-ci est parfois incorrecte, issue probablement de copier-coller faits à partir de la documentation de VB. La fonction *vartype* de VBScript ne fait qu'une partie de ce qui est annoncé ci-dessus.

Le programme précédent, réécrit pour *vartype* donne les résultats suivants :

Programme

```
1  ' types
2  ' par la suite, var est un variant qui prend différentes valeurs
3
4  ' entier
5    var=1
6    wscript.echo "var=" & var & ",type=" & vartype(var)
7
8  ' chaîne de caractères
9    var="deux"
10   wscript.echo "var=" & var & ",type=" & vartype(var)
11
12 ' booléen
13   var=true
14   wscript.echo "var=" & var & ",type=" & vartype(var)
15
16 ' réel
17   var=4.5
18   wscript.echo "var=" & var & ",type=" & vartype(var)
19
20   ' date
21   var=#10/11/2001#
22   wscript.echo "var=" & var & ",type=" & vartype(var)
23
```

Résultats

```
var=1,type=2
var=deux,type=8
var=Vrai,type=11
var=4,5,type=5
var=11/10/01,type=7
```

2.7 Déclarer les variables utilisées par le script

Nous avons indiqué qu'il n'était pas obligatoire de déclarer les variables utilisées par le script. Dans ce cas, si nous écrivons :

1) somme=4
...
2) somme=**smme**+10

avec une faute de frappe *smme* au lieu de somme dans l'instruction 2, vbscript ne signalera aucune erreur. Il supposera que *smme* est une nouvelle variable. Il la créera et dans le contexte de l'instruction 2 l'utilisera en l'initialisant à 0.

Ce genre d'erreurs peut être très difficile à retrouver. Aussi est-il conseillé de forcer la déclaration des variables avec la directive **option explicit** placée en début de script. Ensuite toute variable doit être déclarée avec une instruction **dim** avant sa première utilisation :

option explicit
...
dim somme
1) somme=4
...
2) somme=**smme**+10

Dans cet exemple, vbscript indiquera qu'il y a une variable non déclarée **smme** en 2) comme le montre l'exemple qui suit :

Programme	Résultats

```
1 ' dim
2
3 ' on force la déclaration des variables
4 Option Explicit
5
6 ' qqs instruction
7 Dim somme
8 somme=4
9 somme=smme+10
```

```
dim1.vbs(9, 1) Erreur d'exécution
Microsoft VBScript: Variable non
définie: 'smme'
```

Si dans les courts exemples du document, les variables ne sont la plupart du temps pas déclarées, nous forcerons leur déclaration dès que nous écrirons les premiers scripts significatifs. La directive **Option explicit** sera alors utilisée systématiquement.

2.8 Les fonctions de conversion

Vbscript transforme les données des variants en chaînes, nombres, booléens, ... selon le contexte. La plupart du temps, cela fonctionne bien mais parfois cela donne quelques surprises comme nous le verrons ultérieurement. On peut alors vouloir "forcer" le type de donnée du variant. VBscript possède des fonctions de conversion qui transforment une expression en divers types de données. En voici quelques unes :

Cint (expression)	transforme expression en entier court (integer)
Clng (expression)	transforme expression en entier long (long)
Cdbl (expression)	transforme expression en réel double (double)
Csng (expression)	transforme expression en réel simple (single)
Ccur (expression)	transforme expression en donnée monétaire (currency)

Voici quelques exemples :

Programme

```
' conversions

Dim var

' conversions classiques

' chaîne --> integer
var="4"
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)
var=cint(var)
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)

' chaîne --> long
var="1000000"
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)
var=clng(var)
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)

' chaîne --> double
var="3,4e-5"
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)
var=cdbl(var)
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)

' chaîne --> single
var="3,4e-5"
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)
var=csng(var)
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)

' chaîne --> currency
var="1000,45"
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)
var=ccur(var)
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)

' nombre quelconque --> chaîne
var=14
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)
var="" & var
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)

' double --> integer
var=1000.45
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)
var=cint(var)
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)

var=1000.75
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)
var=cint(var)
wscript.echo "var=" & var & ",code type=" & vartype(var) & ",nom type=" & typename(var)

```

Résultats

```
var=4,code type=8,nom type=String
var=4,code type=2,nom type=Integer
var=1000000,code type=8,nom type=String
var=1000000,code type=3,nom type=Long
var=3,4e-5,code type=8,nom type=String
var=0,000034,code type=5,nom type=Double
var=3,4e-5,code type=8,nom type=String
var=0,000034,code type=4,nom type=Single
var=1000,45,code type=8,nom type=String
var=1000,45,code type=6,nom type=Currency
var=14,code type=2,nom type=Integer
var=14,code type=8,nom type=String
var=1000,45,code type=5,nom type=Double
var=1000,code type=2,nom type=Integer
var=1000,75,code type=5,nom type=Double
var=1001,code type=2,nom type=Integer
```

2.9 Lire des données tapées au clavier

L'objet wscript permet à un script de récupérer des données tapées au clavier. La méthode *wscript.stdin.readLine* permet de lire une ligne de texte tapée au clavier et validée par la touche "Entrée". Cette ligne lue peut être affectée à une variable.

Programme

```
' lectures clavier
wscript.stdout.write "Tapez votre nom : "
nom=wscript.stdin.readLine
wscript.stdout.writeLine "Bonjour " & nom

```

Résultats

```
Tapez votre nom : st
Bonjour st
```

Commentaires :

- Dans la colonne des résultats et dans la ligne [`Tapez votre nom : st`] , *st* est la ligne tapée par l'utilisateur.

Si le texte tapé au clavier représente un nombre, il est quand même considéré avant tout comme une chaîne de caractères comme le montre l'exemple ci-dessous :

Programme	Résultats

```
1 ' lectures clavier
2 wscript.stdout.write "Tapez un nombre : "
3 nombre=wscript.stdin.readLine
4 wscript.stdout.writeLine "nombre lu=" & nombre _
5   & ",type=" & typename(nombre)
6
7
```

```
Tapez un nombre : 14
nombre lu=14,type=String
```

Si ce nombre intervient dans une opération arithmétique, VBscript fera automatiquement la conversion de la chaîne vers un nombre mais pas toujours. Regardons l'exemple qui suit :

Programme	Résultats

```
1 ' lire deux nombres tapés au clavier
2 wscript.stdout.write "nombre1 : "
3 nombre1=wscript.stdin.readLine
4 wscript.stdout.write "nombre2 : "
5 nombre2=wscript.stdin.readLine
6
7 ' calculs
8 somme=nombre1+nombre2
9 diff=nombre1-nombre2
10 produit=nombre1*nombre2
11 quotient=nombre1/nombre2
12
13 ' écrire les résultats à l'écran
14 wscript.echo nombre1 & "+" & nombre2 & "=" & somme
15 wscript.echo nombre1 & "-" & nombre2 & "=" & diff
16 wscript.echo nombre1 & "x" & nombre2 & "=" & produit
17 wscript.echo nombre1 & "/" & nombre2 & "=" & quotient
18
```

```
nombre1 : 3
nombre2 : 4
3+4=34
3-4=-1
3x4=12
3/4=0,75
```

Dans les résultats, on voit que la ligne 8 du script ne s'est pas déroulée comme attendu, ceci parce que (malheureusement) en vbscript l'opérateur + a deux significations : addition de deux nombres ou concaténation de deux chaînes (les deux chaînes sont collées l'une à l'autre). Nous avons vu précédemment que les nombres tapés au clavier étaient lus comme étant des chaînes de caractères et que vbscript transformait celles-ci en nombres selon les besoins. Il l'a correctement fait pour les opérations -,*,/ qui ne peuvent faire intervenir que des nombres mais pas pour l'opérateur + qui lui peut également faire intervenir des chaînes. Il a supposé ici qu'on voulait faire une concaténation de chaînes.

Une solution simple à ce problème est de transformer en nombres les chaînes dès leur lecture comme le montre l'amélioration qui suit du programme précédent :

Programme	Résultats

```vbscript
' lire deux nombres tapés au clavier

' nombre1
wscript.stdout.write "nombre1 : "
nombre1=wscript.stdin.readLine
wscript.echo "nombre1=" & nombre1 & ",type=" & typename(nombre1)
nombre1=clng(nombre1)
wscript.echo "nombre1=" & nombre1 & ",type=" & typename(nombre1)

' nombre2
wscript.stdout.write "nombre2 : "
nombre2=wscript.stdin.readLine
wscript.echo "nombre2=" & nombre2 & ",type=" & typename(nombre2)
nombre2=clng(nombre2)
wscript.echo "nombre2=" & nombre2 & ",type=" & typename(nombre2)

' calculs
somme=nombre1+nombre2
diff=nombre1-nombre2
produit=nombre1*nombre2
quotient=nombre1/nombre2

' écrire les résultats à l'écran
wscript.echo nombre1 & "+" & nombre2 & "=" & somme
wscript.echo nombre1 & "-" & nombre2 & "=" & diff
wscript.echo nombre1 & "x" & nombre2 & "=" & produit
wscript.echo nombre1 & "/" & nombre2 & "=" & quotient
```

```
nombre1 : 3
nombre1=3,type=String
nombre1=3,type=Long
nombre2 : 4
nombre2=4,type=String
nombre2=4,type=Long
3+4=7
3-4=-1
3x4=12
3/4=0,75
```

2.10 Saisir des données avec la fonction inputbox

On peut vouloir saisir des données dans une interface graphique plutôt qu'au clavier. On utilise alors la fonction *inputBox*. Celle-ci admet de nombreux paramètres dont seuls les deux premiers sont fréquemment utilisés :

reponse=inputBox(message,titre)

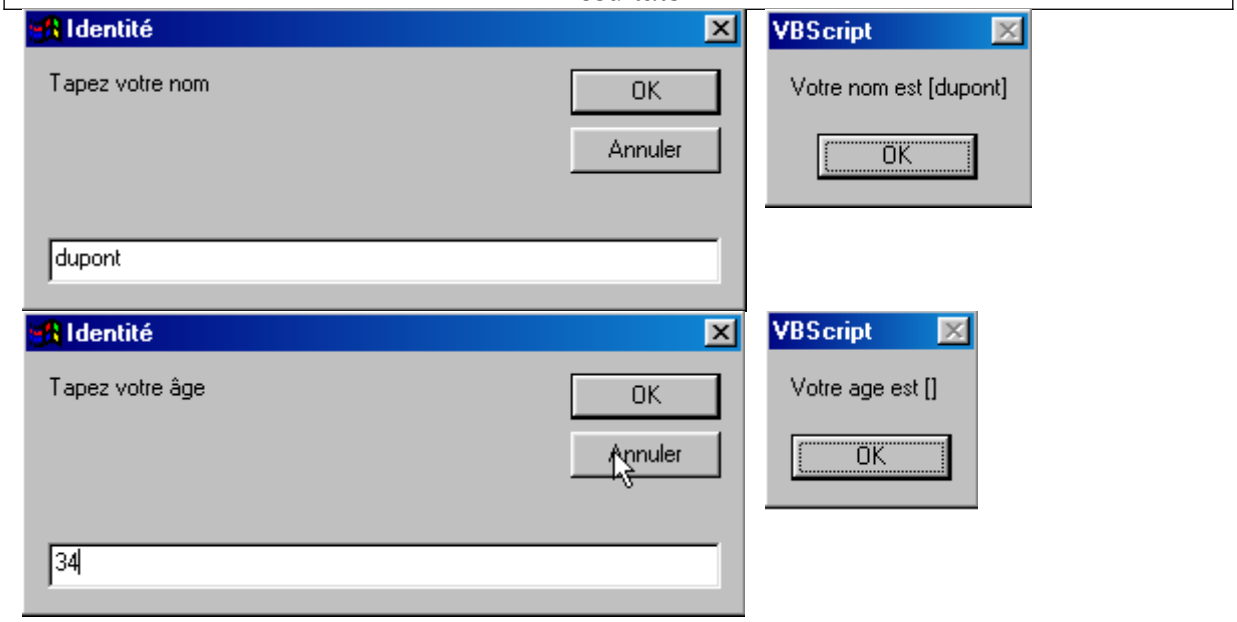
- **message** : la question que vous posez à l'utilisateur
- **titre** (facultatif) : le titre que vous donnez à la fenêtre de saisie
- **reponse** : le texte tapé par l'utilisateur. Si celui-ci a fermé la fenêtre sans répondre, **reponse** est la chaîne vide.

Voici un exemple où on demande le nom et l'âge d'une personne. Pour le nom on donne une information et on fait OK. Pour l'âge, on donne également une information mais on fait Annuler.

Programme

```vbscript
' inputbox

nom=inputBox("Tapez votre nom","Identité")
msgbox "Votre nom est [" & nom & "]"
age=inputBox("Tapez votre âge","Identité")
msgbox "Votre age est [" & age & "]"
```

Résultats



2.11 Utiliser des objets structurés

Il est possible de créer avec vbscript des objets ayant des méthodes et des propriétés. Pour ne pas compliquer les choses, nous allons présenter ici un objet avec des propriétés et pas de méthodes. Considérons une personne. Elle a de nombreuses propriétés qui la caractérisent : taille, poids, couleur de peau, des yeux, des cheveux, ... Nous n'en retiendrons que deux : son nom et son âge. Avant de pouvoir utiliser des objets, il faut créer le moule qui va permettre de les fabriquer. Cela se fait en vbscript avec une **classe**. La classe *personne* pourrait être définie comme suit :

```
class personne
  Dim nom,age
End class
```

C'est l'instruction [`Dim nom,age`] qui définit les deux propriétés de la classe *personne*. Pour créer des exemplaires (on parle d'instances) de la classe *personne*, on écrit :

```
set personne1=new personne
set personne2=new personne
```

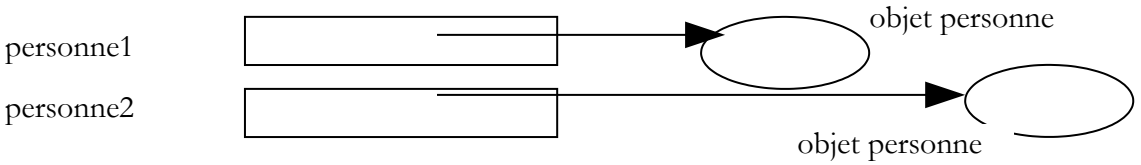
Pourquoi ne pas écrire

```
personne1=new personne
personne2=new personne
```

Parce qu'un variant ne peut contenir un objet. Il peut seulement en contenir l'adresse. En écrivant *set personne1=new personne*, la séquence d'événements suivante prend place :

1. un objet *personne* est créé. Cela veut dire que de la mémoire lui est allouée.
2. l'adresse de cet objet *personne* est affectée à la variable *personne1*

Nous avons alors le schéma mémoire suivant pour les variables *personne1* et *personne2* :



Par abus de langage, on pourra dire que *personne1* est un objet *personne*. On peut accepter cet abus de langage si on se rapelle que *personne1* est en fait l'adresse d'un objet *personne* et non l'objet *personne* lui-même.

Nous avons dit qu'un objet *personne* avait deux propriétés *nom* et *age*. Comment exploiter ces propriétés ? Par la notation *objet.propriété* comme il a été expliqué un peu plus haut. Ainsi

personne1.nom désigne le nom de la personne 1 et *personne1.age* son âge. Voici un court programme d'illustration :

Programme	Résultats

```
1 ' une classe
2 class personne
3   Dim nom
4   Dim age
5 End class
6
7 ' création d'un objet personne
8 Set p1=new personne
9 p1.nom="dupont"
10 p1.age=18
11
12 ' affichage propriétés personne p1
13 wscript.echo "p1=(" & p1.nom & "," & p1.age & ")"
14
```

```
p1=(dupont,18)
```

Le programme précédent pourrait être modifié comme suit :

Programme	Résultats

```
1 ' une classe
2 class personne
3   Dim nom
4   Dim age
5 End class
6
7 ' création d'un objet personne
8 Set p1=new personne
9 With p1
10   .nom="dupont"
11   .age=18
12 End With
13
14 ' affichage propriétés personne p1
15 With p1
16   wscript.echo "nom=" & .nom
17   wscript.echo "age=" & .age
18 End With
19
```

```
nom=dupont
age=18
```

Nous avons utilisé ici la structure *with ... end with* qui permet de "factoriser" des noms d'objets dans des expressions. La structure *with p1 ... end with* des lignes 9-12 et 15-18 permet d'utiliser ensuite la syntaxe *.nom* en lieu et place de *p1.nom* et *.age* en lieu et place de *p1.age*. Cela permet d'alléger l'écriture des instructions où le même nom d'objet est utilisée de façon répétée.

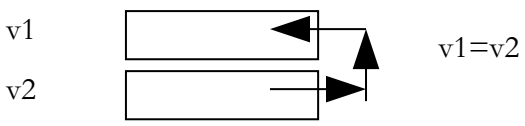
2.12 Affecter une valeur à une variable

Il y a deux instructions pour affecter une valeur à une variable :

1. **variable=expression**
2. **set variable=expression**

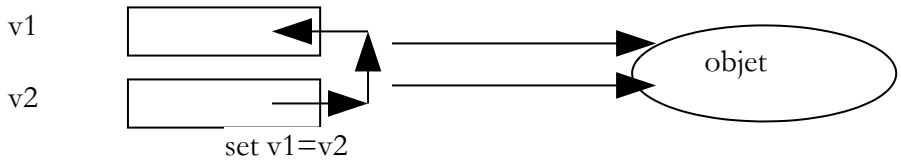
La forme 2 est réservée aux expressions dont le résultat est une référence d'objet. Pour tous les autres types d'expressions c'est la forme 1 qui convient. La différence entre les deux formes est la suivante :

1. dans l'instruction *variable=expression*, variable reçoit une valeur. Si v1 et v2 sont deux variables, écrire v1=v2 affecte la valeur de v1 à v2. On a donc la duplication d'une valeur à deux endroits différents. Si par la suite, la valeur de v2 est modifiée, celle de v1 ne l'est en rien.



2. dans l'instruction *set variable=expression*, variable reçoit comme valeur l'adresse d'un objet. Si v1 et v2 sont deux variables et si v2 est l'adresse d'un objet obj2, écrire set v1=v2

affecte la valeur de v1 à v2, donc l'adresse de l'objet obj2. Lorsque le script manipule ensuite v1 et v2, ce ne sont pas les "valeurs" de v1 et v2 qui sont manipulées mais bien les objets "pointés" par v1 et v2, donc le même objet ici. On dit que v1 et v2 sont deux références au même objet et manipuler ce dernier via v1 ou v2 ne fait aucune différence. Dit autrement, modifier l'objet référencé par v2 modifie celui référencé par v1.



Voici un exemple :

Programme

```
1 ' affectation d'une valeur à une variable
2 i=4
3 j=i
4 wscript.echo "i=" & i
5 wscript.echo "j=" & j
6 ' on modifie j - aucune influence sur i
7 j=j+1
8 wscript.echo "i=" & i
9 wscript.echo "j=" & j
10
11 ' définition d'une classe personne
12 class personne
13   Dim nom,age
14 End class
15
16 ' création d'un e 1er objet personne
17 Set p1=new personne
18 p1.nom="dupont"
19 p1.age=18
20 wscript.echo "p1.nom=" & p1.nom
21 wscript.echo "p1.age=" & p1.age
22 ' p2 va référencer le même objet que p1 et le modifier
23 Set p2=p1
24 p2.age=19
25 ' les deux références p1 et p2 doivent voir les mêmes modifications
26 wscript.echo "p1.nom=" & p1.nom
27 wscript.echo "p1.age=" & p1.age
28 wscript.echo "p2.nom=" & p2.nom
29 wscript.echo "p2.age=" & p2.age
30
```

Résultats

```
i=4
j=4
i=4
j=5
p1.nom=dupont
p1.age=18
p1.nom=dupont
p1.age=19
p2.nom=dupont
p2.age=19
```

2.13 Évaluer des expressions

Les principaux opérateurs permettant d'évaluer des expressions sont les suivantes :

Type d'opérateurs	Opérateurs	Exemple
Arithmétique	+,-,*,/	
	mod	*a mod b* donne le reste de la division entière de a par b. Auparavant a et b ont été transformés en entiers si besoin était.
	\	*a* \ *b* donne le quotient de la division entière de a par b. Auparavant a et b ont été transformés en entiers si besoin était.
	^	*a^b* élève a à la puissance b. Ainsi a^2 est égal à a^2

Comparaison	<,<= >, >= =,<>	*a<>b* est vrai si a est différent de b *a=b* est vrai si a est égal à b a et b peuvent être tous deux des nombres ou tous deux des chaînes de caractères. Dans ce dernier cas, chaine1<chaine2 si dans l'ordre alphabétique chaine1 précède chaine2. Dans la comparaison de chaînes, les majuscules précèdent les minuscules dans l'ordre alphabétique.
	is	*obj1 is obj2* est vrai si obj1 et obj2 sont des références sur le même objet.
Logique	and, or, not, xor	Les opérandes sont tous ici booléens. *bool1 or bool2* est vrai si bool1 ou bool2 est vrai *bool1 and bool2* est vrai si bool1 et bool2 sont vrais *not bool1* est vrai si bool1 est faux et vice-versa *bool1 xor bool2* est vrai si seulement un seul des booléens bool1, bool2 est vrai
Concaténation	&, +	Il est déconseillé d'utiliser l'opérateur + pour concaténer deux chaînes à cause de la confusion possible avec l'addition de deux nombres. On utilisera donc exclusivement l'opérateur &.

2.14 Contrôler l'exécution du programme

2.14.1 Exécuter des actions de façon conditionnelle

L'instruction vbscript permettant de faire des actions selon la valeur vraie/fausse d'une condition est la suivante :

if expression then
    action-vrai-1
    action-vrai-2
    ..
else
    action-faux-1
    action-faux-2
    ...
end if

L'expression *expression* est tout d'abord évaluée. Cette expression doit avoir une valeur booléenne. Si elle a la valeur vrai, les actions du *then* sont exécutées sinon ce sont celles du *else* s'il est présent.

Suit un programme présentant différentes variantes du if-then-else :

Programme	Résultats

```vbscript
' tests avec if

' if - then
i=3
If i>0 Then
  wscript.echo i & " est plus grand que 0"
End If

' if - then sur 1 ligne
If i>2 Then wscript.echo i & " est plus grand que 2": i=i+1
wscript.echo "i=" & i

' if - then - else
If i<10 Then
  wscript.echo i & " est plus petit que 10" : i=i-1
Else
  wscript.echo i & " est plus grand ou égal à 10" : i=i+1
End If
wscript.echo "i=" & i
```

```
3 est plus grand que 0
3 est plus grand que 2
i=4
4 est plus petit que 10
i=3
```

Commentaires :

- en vbscript, on peut écrire *instruction1:instruction2:... : instructionn* au lieu d'écrire une instruction par ligne. C'est cette possibilité qui a été exploitée en ligne 10 par exemple.

2.14.2 Exécuter des actions de façon répétée

Boucle à nombre d'itérations connu

```
for i=idébut to ifin step ipas
        actions
next
```

1. la variable i est ici appelée variable de boucle. Elle peut porter un nom quelconque
2. i prend la valeur idébut
3. la valeur de i est comparée à ifin. Si i<=ifin, les actions situées entre le for... next sont exécutées
4. i est incrémenté de la quantité ipas (i=i+ipas)
5. on reboucle à l'étape 3 précédente. Au bout d'un nombre fini d'étapes, la valeur de i dépassera ifin. L'exécution du script se poursuit avec l'instruction qui suit le next
6. si l'incrément ipas est négatif, la condition de l'étape 3 est changée. On exécute les actions du for...next que si i>=ifin.

On peut sortir d'une boucle for à tout moment avec l'instruction **exit for**.

Boucle à nombre d'itérations inconnu

```
do while condition
        actions
loop
```

1. l'expression condition est évaluée. Si elle est vraie, les actions du while...loop sont exécutées
2. les actions exécutées ont pu modifier la valeur de condition. On reboucle sur l'étape 1 précédente.
3. lorsque l'expression condition devient fausse, la boucle est terminée

On peut sortir d'une boucle do while à tout moment avec l'instruction **exit do**.

Le programme ci-dessous illustre ces points :

Programme

```
1 ' boucle for
2
3 ' un tableau
4 tableau=array(10,20,30,40)
5 ' on fait la somme des éléments de tableau
6 somme=0
7 For i=0 To ubound(tableau)
8   ' on incrémente la somme
9   somme=somme+tableau(i)
10  ' suivi
11  wscript.echo "i=" & i & ",tableau(i)=" & tableau(i) & ",somme=" & somme
12 Next
13
14 ' boucle while - on boucle tant que la somme n'a pas dépassé une certaine valeur
15 somme=0
16 i=0
17 Do While i<=ubound(tableau) And somme<40
18  ' on incrémente la somme
19  somme=somme+tableau(i)
20  ' suivi
21  wscript.echo "i=" & i & ",tableau(i)=" & tableau(i) & ",somme=" & somme
22  ' on passe à l'élément suivant du tableau
23  i=i+1
24 Loop
25
26 ' sorties de boucle
27 For i= 1 To 100
28  wscript.echo "i=" & i
29  If i>10 Then Exit For
30 Next
31 i=1
32 Do While i<=100
33  wscript.echo "i=" & i
34  If i>5 Then Exit Do
35  i=i+1
36 Loop
37
```

Résultats

```
i=0,tableau(i)=10,somme=10
i=1,tableau(i)=20,somme=30
i=2,tableau(i)=30,somme=60
i=3,tableau(i)=40,somme=100
i=0,tableau(i)=10,somme=10
i=1,tableau(i)=20,somme=30
i=2,tableau(i)=30,somme=60
i=1
i=2
i=3
i=4
i=5
i=6
i=7
i=8
i=9
i=10
i=11
i=1
i=2
i=3
i=4
i=5
i=6
```

Note : Dans la phase de développement d'un programme, il n'est pas rare qu'un programme "boucle", c.a.d. qu'il ne s'arrête jamais. En général, le programme exécute une boucle dont la condition de sortie ne peut être vérifiée comme par exemple dans l'exemple qui suit :

```
' boucle infinie
i=0
Do While 1=1
  i=i+1
  wscript.echo i
Loop

' une autre du même genre
i=0
Do While true
  i=i+1
  wscript.echo i
Loop
```

Si on exécute le programme précédent, la première boucle ne s'arrêtera jamais d'elle-même. On peut forcer son arrêt en tapant **CTRL-C** au clavier (touche CTRL et touche C enfoncées en même temps).

2.14.3 Terminer l'exécution du programme

L'instruction ***wscript.quit n*** termine l'exécution du programme en renvoyant un code d'erreur égal à n. Sous DOS, ce code d'erreur peut être testé avec l'instruction if ERRORLEVEL n qui est vrai si le code d'erreur renvoyé par le dernier programme exécuté est >=n. Considérons le programme suivant et ses résultats :

```
1 ' illustration de quit
2 wscript.echo "début"
3 ' on quitte avec le code d'erreur 4
4 wscript.quit 4
5 ' cette instruction ne sera jamais exécutée
6 wscript.echo "fin"
```

```
début
```

Juste après l'exécution du programme, on émet les trois commandes DOS suivantes :

```
1. C:\>if ERRORLEVEL 5 echo 5

2. C:\>if ERRORLEVEL 4 echo 4
   4

3. C: >if ERRORLEVEL 3 echo 3
   3
```

La commande DOS 1 teste si le code d'erreur retourné par le programme est >=5. Si oui, elle affiche (echo) 5 sinon rien.
La commande DOS 2 teste si le code d'erreur retourné par le programme est >=4. Si oui, elle affiche 4 sinon rien.
La commande DOS 3 teste si le code d'erreur retourné par le programme est >=3. Si oui, elle affiche 3 sinon rien.

Des résultats affichés, on peut déduire que le code d'erreur retourné par le programme était 4.

2.15 Les tableaux de données dans un variant

Un variant T peut contenir une liste de valeurs. On dit alors que c'est un tableau. Un tableau T possède diverses propriétés :

- on a accès à l'élément i du tableau T par la syntaxe **T(i)** où i est un entier appelé indice entre 0 et n-1 si T a n éléments.
- on peut connaître l'indice du dernier élément du tableau T avec l'expression **ubound(T)**. Le nombre d'éléments du tableau T est alors ubound(T)+1. On appelle souvent ce nombre la taille du tableau.
- un variant T peut être initialisé avec un tableau vide par la syntaxe **T=array()** ou avec une suite d'éléments par la syntaxe **T=array(élément0, élément1, ..., élémentn)**
- on peut ajouter des éléments à un tableau T déjà créé. Pour cela, on utilise l'instruction **redim preserve T(N)** où N est le nouvel indice du dernier élément du tableau T. L'opération est appelée un redimensionnement (redim). Le mot clé **preserve** indique que lors de ce redimensionnement, le contenu actuel du tableau doit être préservé. En l'absence de ce mot clé, T est redimensionné et vidé de ses éléments.
- un élément T(i) du tableau T est de type variant et peut donc contenir n'importe quelle valeur et en particulier un tableau. Dans ce cas, la notation **T(i)(j)** désigne l'élément j du tableau T(i).

Ces diverses propriétés des tableaux sont illustrées par le programme qui suit :

Programme	Résultats

```
1 ' tableaux
2
3 ' on initialise un tableau T1
4 t1=array(1,-4.5,"deux",true,#01/10/2002#)
5 ' on affiche son nombre d'éléments
6 n1=ubound(t1)+1
7 wscript.echo "Le tableau t1 a " & n1 & " éléments"
8 ' on affiche ses valeurs
9 wscript.echo "t1(0)=" & t1(0)
10 wscript.echo "t1(1)=" & t1(1)
11 wscript.echo "t1(2)=" & t1(2)
12 wscript.echo "t1(3)=" & t1(3)
13 wscript.echo "t1(4)=" & t1(4)
14 ' on peut afficher la totalité du tableau
15 wscript.echo "t1=" & join(t1,":")
16
17 ' on rajoute un élément au tableau t1 - cet élément est lui-même un
18 ReDim Preserve t1(5)
19 t1(5)=array(10,20,30)
20 wscript.echo "t1(5)=" & join(t1(5),":")
21 wscript.echo "t1(5)(1)=" & t1(5)(1)
22
23 ' on redimensionne t1 à 3 éléments
24 ReDim Preserve t1(2)
25 wscript.echo "t1=" & join(t1," ~ ")
26
27 ' on redimensionne à 4 éléments sans préservation de l'existant
28 ReDim t1(3)
29 wscript.echo "t1=" & join(t1,":")
30
31
```

```
Le tableau t1 a 5 éléments
t1(0)=1
t1(1)=-4,5
t1(2)=deux
t1(3)=Vrai
t1(4)=10/01/02
t1=1:-4,5:deux:Vrai:10/01/02
t1(5)=10:20:30
t1(5)(1)=20
t1=1 ~ -4,5 ~ deux
t1=:::
```

Commentaires

- on a utilisé ici une fonction appelée **join** explicitée un peu plus loin.

2.16 Les variables tableaux

Il existe en vbscript une autre façon d'utiliser un tableau, c'est d'utiliser une variable tableau. Une telle variable doit alors être obligatoirement déclarée contrairement aux variables scalaires par une instruction **dim**. Diverses déclarations sont possibles :

- **dim tableau(n)** déclare un tableau statique de n+1 éléments numérotés de 0 à n. Ce type de tableau ne peut pas être redimensionné
- **dim tableau()** déclare un tableau dynamique vide. Il devra être redimensionné pour être utilisé par l'instruction redim de la même manière que pour un variant contenant un tableau
- **dim tableau(n,m)** déclare un tableau à 2 dimensions de (n+1)*(m+1) éléments. L'élément (i,j) du tableau est noté *tableau(i,j)*. On notera la différence avec un variant où le même élément aurait été noté *tableau(i)(j)*.

Pourquoi deux types de tableaux qui finalement sont très proches ? La documentation de vbscript n'en parle pas et n'indique pas non plus si l'un est plus performant que l'autre. Par la suite, nous utiliserons quasi exclusivement le tableau dans un variant dans nos exemples. On se rappellera cependant que VBscript dérive du langage Visual Basic qui contient lui des données typées (integer, double, boolean, ...). Dans ce cas, si on doit utiliser un tableau de nombres réels par exemple, la variable tableau sera plus performante que la variable variant. On déclarera alors quelque chose comme *dim tableau(1000) as double* pour déclarer un tableau de nombres réels ou simplement *dim tableau() as double* si le tableau est dynamique.

Voici un exemple illustrant l'utilisation de variables tableau :

Programme	Résultats

```vbscript
' tableau

' on initialise un tableau T1 fixe
Dim t1(4)
t1(0)=1:t1(1)=-4.5:t1(2)="deux":t1(3)=true:t1(4)=#01/10/2002#

' on affiche son nombre d'éléments
n1=ubound(t1)+1
wscript.echo "Le tableau t1 a " & n1 & " éléments"

' on affiche ses valeurs
wscript.echo "t1(0)=" & t1(0)
wscript.echo "t1(1)=" & t1(1)
wscript.echo "t1(2)=" & t1(2)
wscript.echo "t1(3)=" & t1(3)
wscript.echo "t1(4)=" & t1(4)

' on peut afficher la totalité du tableau
wscript.echo "t1=" & join(t1,":")

' on utilise un tableau t2 dynamique
Dim t2()

' On affiche son nombre d'éléments
n1=ubound(t1)+1
wscript.echo "Le tableau t1 a " & n1 & " éléments"

' on redimensionne t2
ReDim Preserve t2(2)
t2(0)=0
t2(1)=array(10,20,30)
wscript.echo "t2(0)=" & t2(0)
wscript.echo "t2(1)(2)=" & t2(1)(2)

' on redimensionne t2 à 4 éléments sans préservation de l'existant
ReDim t2(3)
wscript.echo "t2=" & join(t2,":")

' tableau t3 à 2 dimensions
Dim t3(1,1)
t3(0,0)=0:t3(0,1)=1:t3(1,0)=10:t3(1,1)=11
wscript.echo "t3(0,0)=" & t3(0,0)
wscript.echo "t3(0,1)=" & t3(0,1)
wscript.echo "t3(1,0)=" & t3(1,0)
wscript.echo "t3(1,1)=" & t3(1,1)

```

```
Le tableau t1 a 5 éléments
t1(0)=1
t1(1)=-4,5
t1(2)=deux
t1(3)=Vrai
t1(4)=10/01/02
t1=1:-4,5:deux:Vrai:10/01/02
Le tableau t1 a 5 éléments
t2(0)=0
t2(1)(2)=30
t2=:::
t3(0,0)=0
t3(0,1)=1
t3(1,0)=10
t3(1,1)=11
```

2.17 Les fonctions split et join

Les fonctions split et join permettent de passer d'une chaîne de caractères à un tableau et vice-versa :

- Si **T** est un tableau et **car** une chaîne de caractères, **join(T,car)** est une chaîne de caractères formée par la réunion de tous les éléments du tableau T, chacun étant séparé du suivant par la chaîne **car**. Ainsi join(array(1,2,3),"abcd") donnera la chaîne "1abcd2abcd3"
- Si **C** est une chaîne de caractères formée d'une suite de champs séparés par la chaîne **car** la fonction **split(C,car)** est un tableau dont les éléments sont les différents de la chaîne C. Ainsi split("1abcd2abcd3","abcd") donnera le tableau (1,2,3)

Voici un exemple :

Programme	Résultats

```vbscript
' transformation tableau-->chaine et vice-versa

' tableau --> chaine
tableau=array("un",2,"trois")
chaine=join(tableau,",")
wscript.echo chaine

' chaine --> tableau
tableau2=split(chaine,",")
For i=0 To ubound(tableau2)
  wscript.echo tableau2(i)
```

```
un,2,trois
un
2
trois
```

```
Next
```

2.18 Les dictionnaires

On a accès à l'élément d'un tableau T lorsqu'on connaît son numéro i. Il est alors accessible par la notation T(i). Il existe des tableaux dont on accèe aux éléments, non pas par un numéro mais par une chaîne de caractères. L'exemple typique de ce type de tableau est le dictionnaire. Lorsqu'on cherche la signification d'un mot dans le "Larousse" ou "Le petit Robert", on accède à celle-ci par le mot. On pourrait représenter ce dictionnaire par un tableau à 2 colonnes :

mot1	description1
mot2	description2
mot3	description3
....	

On pourrait alors écrire des choses comme :

dictionnaire("mot1")="description1"
dictionnaire("mot2")="description2"
...

On est alors proche du fonctionnement d'un tableau si ce n'est que les indices du tableau ne sont pas des nombres entiers mais des chaînes de caractères. On appelle ce type de tableau un **dictionnaire** (ou tableau associatif, hashtable) et les indices chaînes de caractères les **clés** du dictionnaire (keys). L'usage des dictionnaires est extrêmement fréquent dans le monde informatique. Nous avons tous une carte de sécurité sociale avec dessus un numéro. Ce numéro nous identifie de façon unique et donne accès aux informations qui nous concernent. Dans le modèle dictionnaire("clé")="informations", "clé" serait ici le n° de sécurité sociale et "informations" toutes les informations stockées à notre sujet sur les ordinateurs de la sécurité sociale.

Sous Windows, on dispose d'un objet Active X appelé "**Scripting.Dictionary**" qui permet de créer et gérer des dictionnaires. Un objet Active X est un composant logiciel qui expose une interface utilisable par des programmes qui peuvent être écrits en différents langages, tant qu'ils respectent la norme d'utilisation des objets Active X. L'objet Scripting.dictionary est donc utilisable par les langages de programmation de Windows : javascript, per, python, C, C++, vb, vba,... et pas seulement par vbscript.

1 Un objet Scripting.Dictionary est créé par une instruction

```
set dico=wscript.CreateObject("Scripting.Dictionary")
```

ou simplement

```
set dico=CreateObject("Scripting.Dictionary")
```

CreateObject est une méthode de l'objet WScript permettant de créer des instances d'objets Active X. La version 2 montre que wscript peut être un objet implicite. Lorsqu'une méthode ne peut être "rapprochée" d'un objet, le conteneur WSH essaiera de le rapprocher de l'objet wscript.

2 Une fois le dictionnaire créé, on va pouvoir lui ajouter des éléments avec la méthode add :

```
dico.add "clé",valeur
```

va créer une nouvelle entrée dans le dictionnaire associée à la clé "clé". La valeur associée est un variant dont une donnée quelconque.

3 Pour récupérer la valeur associée à une clé donnée on utilise la méthode **item** du

dictionnaire :

var=dico.item("clé")

ou **set var=dico.item("clé)** si la valeur associée à la clé est un objet.

4 L'ensemble des clés du dictionnaire peut être récupéré dans un tableau variant grâce à la méthode **keys** :

clés=dico.keys

clés est un tableau dont on peut parcourir les éléments.

5 L'ensemble des valeurs du dictionnaire peut être récupéré dans un tableau variant grâce à la méthode **items** :

valeurs=dico.items

items est un tableau dont on peut parcourir les éléments.

6 L'existence d'une clé peut être testée avec la méthode **exists** :

dico.exists("clé") est vrai si la clé "clé" existe dans le dictionnaire

7 On peut enlever une entrée du dictionnaire (clé+valeur) avec la méthode **remove** :

dico.remove("clé") enlève l'entrée du dictionnaire associée à la clé "clé". **dico.removeall** enlève toutes les clés, c.a.d. vide le dictionnaire.

Le programme suivant utilise ces diverses possibilités :

Programme

```
' création et utilisation d'un dictionnaire
Set dico=CreateObject("Scripting.Dictionary")

' remplisage dico
dico.add "clé1","valeur1"
dico.add "clé2","valeur2"
dico.add "clé3","valeur3"

' nombre d'éléments
wscript.echo "Le dictionnaire a " & dico.count & " éléments"

' liste des clés
wscript.echo "liste des clés"
cles=dico.keys
For i=0 To ubound(cles)
  wscript.echo cles(i)
Next

' liste des valeurs
wscript.echo "liste des valeurs"
valeurs=dico.items
For i=0 To ubound(valeurs)
  wscript.echo valeurs(i)
Next

' liste des clés et valeurs
wscript.echo "liste des clés et valeurs"
cles=dico.keys
For i=0 To ubound(cles)
  wscript.echo "dico(" & cles(i) & ")=" & dico.item(cles(i))
Next

' recherche d'éléments
' clé1
If dico.exists("clé1") Then
    wscript.echo "La clé clé1 existe dans le dictionnaire et la valeur associée est " &
dico.item("clé1")
  Else
    wscript.echo "La clé clé1  n'existe pas dans le dictionnaire"
  End If
' clé4
If dico.exists("clé4") Then
    wscript.echo "La clé clé4 existe dans le dictionnaire et la valeur associée est " &
dico.item("clé4")
  Else
```

```
    wscript.echo "La clé clé4  n'existe pas dans le dictionnaire"
  End If

' on enlève la clé 1
dico.remove("clé1")

' liste des clés et valeurs
wscript.echo "liste des clés et valeurs après suppression de clé1"
cles=dico.keys
For i=0 To ubound(cles)
  wscript.echo "dico(" & cles(i) & ")=" & dico.item(cles(i))
Next

' on supprime tout
dico.removeall

' liste des clés et valeurs
wscript.echo "liste des clés et valeurs après suppression de tous les éléments"
cles=dico.keys
For i=0 To ubound(cles)
  wscript.echo "dico(" & cles(i) & ")=" & dico.item(cles(i))
Next

' fin
wscript.quit 0
```

Résultats

```
Le dictionnaire a 3 éléments
liste des clés
clé1
clé2
clé3
liste des valeurs
valeur1
valeur2
valeur3
liste des clés et valeurs
dico(clé1)=valeur1
dico(clé2)=valeur2
dico(clé3)=valeur3
La clé clé1 existe dans le dictionnaire et la valeur associée est valeur1
La clé clé4  n'existe pas dans le dictionnaire
liste des clés et valeurs après suppression de clé1
dico(clé2)=valeur2
dico(clé3)=valeur3
liste des clés et valeurs après suppression de tous les éléments
```

2.19 Trier un tableau ou un dictionnaire

Il est courant de vouloir trier un tableau ou un dictionnaire dans l'ordre croissant ou décroissant de ses valeurs ou de ses clés pour un dictionnaire. Alors que dans la pupart des langages, existent des fonctions de tri, il ne semble pas en exister en vbscript. C'est une lacune.

2.20 Les arguments d'un programme

Il est possible d'appeler un programme vbscript en lui passant des paramètres comme dans :

```
cscript prog1.vbs arg1 arg2 .... argn
```

Cela permet à l'utilisateur de passer des informations au programme. Comment celui-ci fait-il pour les récupérer ? Regardons le programme suivant :

Programme	Résultats

```
' arguments

Dim arguments, I

' on récupère l'objet Arguments de l'objet WScript
' l'objet Arguments est de type collection
Set arguments = WScript.Arguments
' on affiche le nombre d'arguments
wscript.echo "Il y a " & arguments.count & " arguments"
' on affiche les arguments eux-mêmes
For I = 0 To arguments.Count - 1
   WScript.Echo arguments(I)
Next
```

```
C:\>cscript arg1.vbs a b c

Il y a 3 arguments
a
b
c
```

Commentaires

- *WScript.Arguments* est la collection des arguments passés au script
- une collection C est un objet qui a
  - une propriété *count* qui est le nombre d'éléments dans la collection
  - une méthode *C(i)* qui donne l'élément i de la collection

2.21 Une première application : IMPOTS

On se propose d'écrire un programme permettant de calculer l'impôt d'un contribuable. On se place dans le cas simplifié d'un contribuable n'ayant que son seul salaire à déclarer :

- on calcule le nombre de parts du salarié **nbParts=nbEnfants/2 +1** s'il n'est pas marié, **nbEnfants/2+2** s'il est marié, où *nbEnfants* est son nombre d'enfants.
- on calcule son revenu imposable **R=0.72*S** où S est son salaire annuel
- on calcule son coefficient familial **Q=R/N**

on calcule son impôt I d'après les données suivantes

```
12620.0     0       0
13190       0.05    631
15640       0.1     1290.5
24740       0.15    2072.5
31810       0.2     3309.5
39970       0.25    4900
48360       0.3     6898.5
55790       0.35    9316.5
92970       0.4     12106
127860      0.45    16754.5
151250      0.50    23147.5
172040      0.55    30710
195000      0.60    39312
0           0.65    49062
```

Chaque ligne a 3 champs. Pour calculer l'impôt I, on recherche la première ligne où *QF<=champ1*. Par exemple, si QF=30000 on trouvera la ligne

24740 0.15 2072.5

L'impôt I est alors égal à **0.15*R - 2072.5*nbParts**. Si QF est tel que la relation *QF<=champ1* n'est jamais vérifiée, alors ce sont les coefficcients de la dernière ligne qui sont utilisés. Ici :

0 0.65 49062

ce qui donne l'impôt I=0.65*R - 49062*nbParts.

Le programme est le suivant :

Programme

```
' calcul de l'impôt d'un contribuable
' le programme doit être appelé avec trois paramètres : marié enfants salaire
' marié : caractère O si marié, N si non marié
' enfants : nombre d'enfants
```

```
' salaire : salaire annuel sans les centimes

' aucune vérification de la validité des données n'est faite mais on
' vérifie qu'il y en a bien trois

' déclaration obligatoire des variables
Option Explicit

' on vérifie qu'il y a 3 arguments
  Dim nbArguments
  nbArguments=wscript.arguments.count
  If nbArguments<>3 Then
    wscript.echo "Syntaxe : pg marié enfants salaire"
    wscript.echo "marié : caractère O si marié, N si non marié"
    wscript.echo "enfants : nombre d'enfants"
    wscript.echo "salaire : salaire annuel sans les centimes"
    ' arrêt avec code d'erreur 1
    wscript.quit 1
  End If

' on récupère les arguments sans vérifier leur validité
  Dim marie, enfants, salaire
  If wscript.arguments(0) = "O" Or wscript.arguments(0)="o" Then
    marie=true
  Else
    marie=false
  End If
  ' enfants est un nombre entier
  enfants=cint(wscript.arguments(1))
  ' salaire est un entier long
  salaire=clng(wscript.arguments(2))

  ' on définit les données nécessaire au calcul de l'impôt dans 3 tableaux
  Dim limites, coeffn, coeffr
  limites=array(12620,13190,15640,24740,31810,39970,48360, _
    55790,92970,127860,151250,172040,195000,0)
  coeffr=array(0,0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45, _
    0.5,0.55,0.6,0.65)
  coeffn=array(0,631,1290.5,2072.5,3309.5,4900,6898.5,9316.5, _
    12106,16754.5,23147.5,30710,39312,49062)

  ' on calcule le nombre de parts
  Dim nbParts
  If marie=true Then
    nbParts=(enfants/2)+2
  Else
    nbParts=(enfants/2)+1
  End If
  If enfants>=3 Then nbParts=nbParts+0.5

  ' on calcule le quotient familial et le revenu imposable
  Dim revenu, qf
  revenu=0.72*salaire
  qf=revenu/nbParts

  ' on calcule l'impôt
  Dim i, impot
  i=0
  Do While i<ubound(limites) And qf>limites(i)
    i=i+1
  Loop
  impot=int(revenu*coeffr(i)-nbParts*coeffn(i))

  ' on affiche le résultat
  wscript.echo "impôt=" & impot

  ' on quitte sans erreur
  wscript.quit 0
```

Résultats

```
C:\>cscript impots1.vbs o 2 200000

impôt=22504

C:\>cscript impots1.vbs o 2 20000

impôt=0

C:\>cscript impots1.vbs o 2 2000000

impôt=746064

C:\>cscript impots1.vbs n 2 200000

impôt=33388
```

```
C:\>cscript impots1.vbs n 3 200000

impôt=22504

C:\>cscript impots1.vbs

Syntaxe : pg marié enfants salaire
marié : caractère O si marié, N si non marié
enfants : nombre d'enfants
salaire : salaire annuel sans les centimes
```

Commentaires :

- le programme utilise ce qui a été exposé précédemment (déclaration des variables, arguments, changements de types, tests, boucles, tableau dans un variant)
- il ne vérifie pas la validité des données, ce qui serait anormal dans un programme réel
- seule la boucle while présente une difficulté. Elle cherche à déterminer l'indice i du tableau limites pour lequel on a limites(i)>qf et cela pour i<ubound(limites) (c.a.d. ici i<13) car le dernier élément du tableau limites n'est pas significatif. Il a été ajouté uniquement pour que le test `[Do While i<ubound(limites) And qf>limites(i)]` puisse se faire pour i=13. Le test est alors `13<13 and qf>limites(13)` et il faut alors (en vbscript) que limites(13) existe. Lorsqu'on sort de la boucle while, la dernière valeur de i calculée permet de calculer l'impôt : `[impot=int(revenu*coeffr(i)-nbParts*coeffn(i))]`.

3 La gestion des erreurs

En programmation, il y a une règle absolue : un programme ne doit jamais "planter" sauvagement. Toutes les erreurs qui peuvent se produire lors de l'exécution du programme doivent être gérées et des messages d'erreurs significatifs générés.

Si nous reprenons l'exemple des impôts traité précédemment, que se passe-t-il si l'utilisateur entre n'importe quoi pour le nombre d'enfants. Regardons sur cet exemple :

```
C:\>cscript impots1.vbs o xyzt 200000
C:\impots1.vbs(33, 3) Erreur d'exécution Microsoft VBScript: Type incompatible: 'cint'
```

C'est ce qu'on appelle un plantage sauvage. Il y a eu "plantage" sur l'instruction `enfants=cint(wscript.arguments(1))` car *arguments(1)* contenait la chaîne "xyzt".

Avant d'utiliser un variant dont on ne connaît pas la nature exacte, il faut vérifier son sous-type exact. On peut faire ceci de différentes façons :

- tester le type réel de la donnée contenue dans un variant avec les fonctions *vartype* ou *typename*
- utiliser une expression régulière pour vérifier que le contenu du variant correspond à un certain modèle
- laisser l'erreur se produire puis l'intercepter pour ensuite la gérer

Nous examinons ces différentes méthodes.

3.1 Connaître le type exact d'une donnée

Rappelons que les fonctions *vartype* ou *varname* permettent de connaître le type exact d'une donnée. Cela ne nous est pas toujours d'un grand secours. Par exemple, lorsque nous lisons une donnée tapée au clavier, les fonctions *vartype* et *typename* vont nous dire que c'est une chaîne de caractères car c'est ainsi qu'est considérée toute donnée tapée au clavier. Cela ne nous dit pas si

cette chaîne peut par exemple être considérée comme un nombre valide. On utilise alors d'autres fonctions pour avoir accès à ce type d'informations :

isNumeric(expression)	rend vrai si expression peut être utilisée comme un nombre
isDate(expression)	rend vrai si expression peut être utilisée comme une date
isEmpty(var)	rend vrai si la variable var n'a pas été initialisée
isNull(var)	rend vrai si la variable var contient des données invalides
isArray(var)	rend vrai si var est un tableau
isObject(var)	rend vrai si var est un objet

L'exemple suivant demande de taper une donnée au clavier jusqu'à ce que celle-ci soit reconnue comme un nombre :

Programme

```
' lecture d'une donnée jusqu'à ce que celle-ci soit reconnue comme un nombre

Option Explicit

Dim fini, nombre

' on boucle tant que la donnée saisie n'est pas correcte
' la boucle est contrôlée par un booléen fini, mis à faux au départ (= ce n'est pas
fini)

fini=false
Do While Not fini
  ' on demande le nombre
  wscript.stdout.write "Tapez un nombre : "
  ' on le lit
  nombre=wscript.stdin.readLine
  ' le type est forcément string lors d'une lecture
  wscript.echo "Type de la donnée lue : " & typename(nombre) & "," & vartype(nombre)
  ' on teste le type réel de la donnée lue
  If isNumeric(nombre) Then
    fini=true
  Else
    wscript.echo "Erreur, vous n'avez pas tapé un nombre. Recommencez svp..."
  End If
Loop

' confirmation
wscript.echo "Merci pour le nombre " & nombre

' et fin
wscript.quit 0
```

Résultats

```
Tapez un nombre : a
Type de la donnée lue : String,8
Erreur, vous n'avez pas tapé un nombre. Recommencez svp...
Tapez un nombre : -12
Type de la donnée lue : String,8
Merci pour le nombre -12
```

La fonction *isNumeric* ne nous dit pas si une expression est un entier ou pas. Pour avoir cette information, il faut faire des tests supplémentaires. L'exemple suivant demande un nombre entier >0 :

Programme

```
' lecture d'une donnée jusqu'à ce que celle-ci soit reconnue comme un nombre entier >0

Option Explicit

Dim fini, nombre

' on boucle tant que la donnée saisie n'est pas correcte
```

```
' la boucle est contrôlée par un booléen fini, mis à faux au départ (= ce n'est pas
fini)

fini=false
Do While Not fini
  ' on demande le nombre
  wscript.stdout.write "Tapez un nombre entier >0: "
  ' on le lit
  nombre=wscript.stdin.readLine
  ' on teste le type réel de la donnée lue
  If isNumeric(nombre) Then
    ' est-ce un entier (nombre égal à sa partie entière) positif ?
    If (nombre-int(nombre))=0 And nombre>0 Then
      fini=true
    End If
  End If
  ' msg d'erreur éventuel
  If Not fini Then wscript.echo "Erreur, vous n'avez pas tapé un nombre entier >0.
Recommencez svp..."
Loop

' confirmation
wscript.echo "Merci pour le nombre entier >0 : " & nombre

' et fin
wscript.quit 0
```

Résultats

```
Tapez un nombre entier >0: a
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: -1
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: 10.6
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: 12
Merci pour le nombre entier >0 : 12
```

Commentaires :

- *int(nombre)* donne la partie entière d'un nombre. Un nombre égal à sa partie entière est un entier.
- On notera, de façon intéressante, qu'il a fallu employer le test `If (nombre-int(nombre))=0 And nombre>0` parce que le test `If nombre=int(nombre) And nombre>0` ne donnait pas les résultats escomptés. Il ne détectait pas les nombres entiers positifs. Nous laissons au lecteur le soin de découvrir pourquoi.
- Le test `If (nombre-int(nombre))=0` n'est pas totalement fiable. Regardons l'exemple d'exécution suivant :

```
Tapez un nombre entier >0: 4,000000000000000000000001
Merci pour le nombre entier >0 : 4,000000000000000000000001
```

Les nombres réels ne sont pas représentés de façon exacte mais de façon approchée. Et ici, l'opération *nombre-int(nombre)* a donné 0 à la précision près de l'ordinateur.

3.2 Les expressions régulières

Les expressions régulières nous permettent de tester le format d'une chaîne de caractères. Ainsi on peut vérifier qu'une chaîne représentant une date est au format jj/mm/aa. On utilise pour cela un modèle et on compare la chaîne à ce modèle. Ainsi dans cet exemple, j m et a doivent être des chiffres. Le modèle d'un format de date valide est alors **"\d\d/\d\d/\d\d"** où le symbole **\d** désigne un chiffre. Les symboles utilisables dans un modèle sont les suivants (documentation Microsoft) :

Caractère	Description

\	Marque le caractère suivant comme caractère spécial ou littéral. Par exemple, "n" correspond au caractère "n". "\n" correspond à un caractère de nouvelle ligne. La séquence "\\" correspond à "\", tandis que "\(" correspond à "(".
^	Correspond au début de la saisie.
$	Correspond à la fin de la saisie.
*	Correspond au caractère précédent zéro fois ou plusieurs fois. Ainsi, "zo*" correspond à "z" ou à "zoo".
+	Correspond au caractère précédent une ou plusieurs fois. Ainsi, "zo+" correspond à "zoo", mais pas à "z".
?	Correspond au caractère précédent zéro ou une fois. Par exemple, "a?ve?" correspond à "ve" dans "lever".
.	Correspond à tout caractère unique, sauf le caractère de nouvelle ligne.
(modèle)	Recherche le *modèle* et mémorise la correspondance. La sous-chaîne correspondante peut être extraite de la collection **Matches** obtenue, à l'aide d'Item **[0]...[n]**. Pour trouver des correspondances avec des caractères entre parenthèses ( ), utilisez "\(" ou "\)".
x|y	Correspond soit à *x* soit à *y*. Par exemple, "z|foot" correspond à "z" ou à "foot". "(z|f)oo" correspond à "zoo" ou à "foo".
{*n*}	*n* est un nombre entier non négatif. Correspond exactement à *n* fois le caractère. Par exemple, "o{2}" ne correspond pas à "o" dans "Bob," mais aux deux premiers "o" dans "fooooot".
{*n*,}	*n* est un entier non négatif. Correspond à au moins *n* fois le caractère. Par exemple, "o{2,}" ne correspond pas à "o" dans "Bob", mais à tous les "o" dans "fooooot". "o{1,}" équivaut à "o+" et "o{0,}" équivaut à "o*".
{*n*,*m*}	*m* et *n* sont des entiers non négatifs. Correspond à au moins *n* et à au plus *m* fois le caractère. Par exemple, "o{1,3}" correspond aux trois premiers "o" dans "fooooooot" et "o{0,1}" équivaut à "o?".
[*xyz*]	Jeu de caractères. Correspond à l'un des caractères indiqués. Par exemple, "[abc]" correspond à "a" dans "plat".
[^*xyz*]	Jeu de caractères négatif. Correspond à tout caractère non indiqué. Par exemple, "[^abc]" correspond à "p" dans "plat".
[*a-z*]	Plage de caractères. Correspond à tout caractère dans la série spécifiée. Par exemple, "[a-z]" correspond à tout caractère alphabétique minuscule compris entre "a" et "z".
[^*m-z*]	Plage de caractères négative. Correspond à tout caractère ne se trouvant pas dans la série spécifiée. Par exemple, "[^m-z]" correspond à tout caractère ne se trouvant pas entre "m" et "z".
\b	Correspond à une limite représentant un mot, autrement dit, à la position entre un mot et un espace. Par exemple, "er\b" correspond à "er" dans "lever", mais pas à "er" dans "verbe".
\B	Correspond à une limite ne représentant pas un mot. "en*t\B" correspond à "ent" dans "bien entendu".
\d	Correspond à un caractère représentant un chiffre. Équivaut à [0-9].
\D	Correspond à un caractère ne représentant pas un chiffre. Équivaut à [^0-9].
\f	Correspond à un caractère de saut de page.

\n	Correspond à un caractère de nouvelle ligne.
\r	Correspond à un caractère de retour chariot.
\s	Correspond à tout espace blanc, y compris l'espace, la tabulation, le saut de page, etc. Équivaut à "[\f\n\r\t\v]".
\S	Correspond à tout caractère d'espace non blanc. Équivaut à "[^ \f\n\r\t\v]".
\t	Correspond à un caractère de tabulation.
\v	Correspond à un caractère de tabulation verticale.
\w	Correspond à tout caractère représentant un mot et incluant un trait de soulignement. Équivaut à "[A-Za-z0-9_]".
\W	Correspond à tout caractère ne représentant pas un mot. Équivaut à "[^A-Za-z0-9_]".
****num	Correspond à *num*, où *num* est un entier positif. Fait référence aux correspondances mémorisées. Par exemple, "(.)\1" correspond à deux caractères identiques consécutifs.
*****n*	Correspond à *n*, où *n* est une valeur d'échappement octale. Les valeurs d'échappement octales doivent comprendre 1, 2 ou 3 chiffres. Par exemple, "\11" et "\011" correspondent tous les deux à un caractère de tabulation. "\0011" équivaut à "\001" & "1". Les valeurs d'échappement octales ne doivent pas excéder 256. Si c'était le cas, seuls les deux premiers chiffres seraient pris en compte dans l'expression. Permet d'utiliser les codes ASCII dans des expressions régulières.
\x*n*	Correspond à *n*, où *n* est une valeur d'échappement hexadécimale. Les valeurs d'échappement hexadécimales doivent comprendre deux chiffres obligatoirement. Par exemple, "\x41" correspond à "A". "\x041" équivaut à "\x04" & "1". Permet d'utiliser les codes ASCII dans des expressions régulières.

Un élément dans un modèle peut être présent en 1 ou plusieurs exemplaires. Considérons quelques exemples autour du symbole **\d** qui représente 1 chiffre :

modèle	signification
\d	un chiffre
\d?	0 ou 1 chiffre
\d*	0 ou davantage de chiffres
\d+	1 ou davantage de chiffres
\d{2}	2 chiffres
\d{3,}	au moins 3 chiffres
\d{5,7}	entre 5 et 7 chiffres

Imaginons maintenant le modèle capable de décrire le format attendu pour une chaîne de caractères :

chaîne recherchée	modèle
une date au format jj/mm/aa	\d{2}/\d{2}/\d{2}
une heure au format hh:mm:ss	\d{2}:\d{2}:\d{2}
un nombre entier non signé	\d+
un suite d'espaces éventuellement vide	\s*
un nombre entier non signé qui peut être précédé ou suivi d'espaces	\s*\d+\s*
un nombre entier qui peut être signé et précédé ou suivi d'espaces	\s*[+|-]?\s*\d+\s*
un nombre réel non signé qui peut être précédé ou suivi d'espaces	\s*\d+(.\d*)?\s*

un nombre réel qui peut être signé et précédé ou suivi d'espaces	\s*[+|]?\s*\d+(.\d*)?\s*
une chaîne contenant le mot juste	\bjuste\b

On peut préciser où on recherche le modèle dans la chaîne :

modèle	signification
^modèle	le modèle commence la chaîne
modèle$	le modèle finit la chaîne
^modèle$	le modèle commence et finit la chaîne
modèle	le modèle est cherché partout dans la chaîne en commençant par le début de celle-ci.

chaîne recherchée	modèle
une chaîne se terminant par un point d'exclamation	!$
une chaîne se terminant par un point	\.$
une chaîne commençant par la séquence //	^//
une chaîne ne comportant qu'un mot éventuellement suivi ou précédé d'espaces	^\s*\w+\s*$
une chaîne ne comportant deux mot éventuellement suivis ou précédés d'espaces	^\s*\w+\s*\w+\s*$
une chaîne contenant le mot secret	\bsecret\b

Les sous-ensembles d'un modèle peuvent être "récupérés". Ainsi non seulement, on peut vérifier qu'une chaîne correspond à un modèle particulier mais on peut récupérer dans cette chaîne les éléments correspondant aux sous-ensembles du modèle qui ont été entourés de parenthèses. Ainsi si on analyse une chaîne contenant une date jj/mm/aa et si on veut de plus récupérer les éléments jj, mm, aa de cette date on utilisera le modèle (\d\d)/(\d\d)/(\d\d).

Voyons sur cette exemple, comment on opère avec vbscript.

1. il nous faut tout d'abord créer un objet **RegExp** (Regular Expression)
   set modele=new regexp
2. ensuite on fixe le modèle à tester
   modele.pattern="(\d\d)/(\d\d)/(\d\d)"
3. on peut vouloir ne pas faire de différence entre majuscules et minuscules (par défaut elle est faite). Ici ça n'a aucune importance.
   modele.IgnoreCase=true
4. on peut vouloir rechercher le modèle plusieurs fois dans la chaîne (par défaut ce n'est pas fait)
   modele.Global=true
   Une recherche globale n'a de sens que si le modèle utilisé ne fait pas référence au début ou à la fin de la chaîne.
5. on recherche alors toutes les correspondances du modèle dans la chaîne :
   set correspondances=modele.execute(chaine)
   La méthode *execute* d'un objet *RegExp* rend une collection d'objets de type **match**. Cet objet a une propriété **value** qui est l'élément de *chaine* correspondant au modèle. Si on a écrit *modele.global=true*, on peut avoir plusieurs correspondances. C'est pourquoi le résultat de la méthode *execute* est une collection de correspondances.
6. le nombre de correspondances est donné par **correspondances.count**. Si ce nombre vaut 0, c'est que le modèle n'a été trouvé nulle part. La valeur de la correspondance n° i est donnée par *correspondances(i).value*. Si le modèle contient des sous-modèles entre parenthèses, alors l'éléments de *correspondances(i)* correspondant à la parenthèse j du modèle est **correspondances(i).submatches(j)**.

Tout ceci est montré dans l'exemple qui suit :

Programme
```
' expression régulière

' on veut vérifier qu'une chaîne contient une date au format jj/mm/aa

Option Explicit
Dim modele

' on définit le modèle
Set modele=new regexp
modele.pattern="\b(\d\d)/(\d\d)/(\d\d)\b"  ' une date n'importe où dans la chaîne
modele.global=true                         ' on recherchera le modèle plusieurs fois dans
la chaîne

' c'est l'utilisateur qui donne la chaîne dans laquelle on cherchera le modèle
Dim chaine, correspondances, i

chaine=""
' on boucle tant que chaine<>"fin"
Do While true
  ' on demande à l'utilisateur de taper un texte
  wscript.stdout.writeLine "Tapez un texte contenant des dates au format jj/mm/aa et fin
pour arrêter : "
  chaine=wscript.stdin.readLine
  ' fini si chaine=fin
  If chaine="fin" Then Exit Do
  ' on compare la chaîne lue au modèle de la date
  Set correspondances=modele.execute(chaine)
  ' a-t-on trouvé une correspondance
  If correspondances.count<>0 Then
    ' on a au moins une correspondance
    For i=0 To correspondances.count-1
      ' on affiche la correspondance i
      wscript.echo "J'ai trouvé la date " & correspondances(i).value
      ' on récupère les sous-éléments de la correspondance i
      wscript.echo "Les éléments de la date " & i & " sont (" &
correspondances(i).submatches(0) & ","
      & correspondances(i).submatches(1) & "," & correspondances(i).submatches(2) & ")"
    Next
  Else
    ' pas de correspondance
    wscript.echo "Je n'ai pas trouvé de date au format jj/mm/aa dans votre texte"
  End If
Loop

' fini
wscript.quit 0
```

Résultats

```
Tapez un texte contenant des dates au format jj/mm/aa et fin pour arrÛter :
aujourd'hui on est le 01/01/01 et demain sera le 02/01/02
J'ai trouvé la date 01/01/01
Les éléments de la date 0 sont (01,01,01)
J'ai trouvé la date 02/01/02
Les éléments de la date 1 sont (02,01,02)

Tapez un texte contenant des dates au format jj/mm/aa et fin pour arrÛter :
une date au format incorrect : 01/01/2002
Je n'ai pas trouvé de date au format jj/mm/aa dans votre texte

Tapez un texte contenant des dates au format jj/mm/aa et fin pour arrÛter :
une suite de dates : 10/10/10, 11/11/11, 12/12/12
J'ai trouvé la date 10/10/10
Les éléments de la date 0 sont (10,10,10)
J'ai trouvé la date 11/11/11
Les éléments de la date 1 sont (11,11,11)
J'ai trouvé la date 12/12/12
Les éléments de la date 2 sont (12,12,12)

Tapez un texte contenant des dates au format jj/mm/aa et fin pour arrÛter :
fin
```

Avec les expressions régulières, le programme testant qu'une saisie clavier est bien un nombre entier positif pourrait s'écrire comme suit :

Programme

```
' lecture d'une donnée jusqu'à ce que celle-ci soit reconnue comme un nombre
```

```
Option Explicit

Dim fini, nombre

' on définit le modèle d'un nombre entier positif (mais qui peut être nul)
Dim modele
Set modele=new regexp
modele.pattern="^\s*\d+\s*$"

' on boucle tant que la donnée saisie n'est pas correcte
' la boucle est contrôlée par un booléen fini, mis à faux au départ (= ce n'est pas
fini)

fini=false
Do While Not fini
  ' on demande le nombre
  wscript.stdout.write "Tapez un nombre entier >0: "
  ' on le lit
  nombre=wscript.stdin.readLine
  ' on teste le format de la donnée lue
  Dim correspondances
  Set correspondances=modele.execute(nombre)
  ' le modèle a-t-il été vérifié ?
  If correspondances.count<>0 Then
    ' c'est un entier mais est-il >0 ?
    nombre=cint(nombre)
    If nombre>0 Then
      fini=true
    End If
  End If
  ' msg d'erreur éventuel
  If Not fini Then wscript.echo "Erreur, vous n'avez pas tapé un nombre entier >0.
Recommencez svp..."
Loop

' confirmation
wscript.echo "Merci pour le nombre entier >0 : " & nombre

' et fin
wscript.quit 0
```

Résultats

```
Tapez un nombre entier >0: 10.3
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: abcd
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: -4
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: 0
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: 1
Merci pour le nombre entier >0 : 1
```

Trouver l'expression régulière qui nous permet de vérifier qu'une chaîne correspond bien à un certain modèle est parfois un véritable défi. Le programme suivant permet de s'entraîner. Il demande un modèle et une chaîne et indique alors si la chaîne correspond ou non au modèle.

Programme

```
' expression régulière

' on veut vérifier qu'une chaîne correspond à un modèle

Option Explicit

' on définit le modèle
Dim modele
Set modele=new regexp
modele.global=true                    ' on recherchera le modèle plusieurs fois dans
la chaîne

' c'est l'utilisateur qui donne la chaîne dans laquelle on cherchera le modèle
Dim chaine, correspondances, i

Do While true
  ' on demande à l'utilisateur de taper un modèle
  wscript.stdout.write "Tapez le modèle à tester et fin pour arrêter : "
  modele.pattern=wscript.stdin.readLine
  ' fini ?
  If modele.pattern="fin" Then Exit Do
  ' on demande à l'utilisateur les chaînes à comparer au modèle
  Do While true
    ' on demande à l'utilisateur de taper un modèle
```

```
    wscript.stdout.writeLine "Tapez la chaîne à tester avec le modèle [" &
modele.pattern & "] et fin pour arrêter : "
    chaine=wscript.stdin.readLine
    ' fini ?
    If chaine="fin" Then Exit Do
    ' on compare la chaîne lue au modèle de la date
    Set correspondances=modele.execute(chaine)
    ' a-t-on trouvé une correspondance
    If correspondances.count<>0 Then
      ' on a au moins une correspondance
      For i=0 To correspondances.count-1
        ' on affiche la correspondance i
        wscript.echo "J'ai trouvé la correspondance " & correspondances(i).value
    Next
    Else
      ' pas de correspondance
      wscript.echo "Je n'ai pas trouvé de correspondance"
    End If
  Loop
Loop

' fini
wscript.quit 0
```

Résultats

```
Tapez le modèle à tester et fin pour arrêter : ^\s*\d+(\,\d+)*\s*$

Tapez la chaîne à tester avec le modèle [^\s*\d+(\,\d+)*\s*$] et fin pour arrêter :
18
J'ai trouvé la correspondance [18]

Tapez la chaîne à tester avec le modèle [^\s*\d+(\,\d+)*\s*$] et fin pour arrêter :
   145.678
Je n'ai pas trouvé de correspondance

Tapez la chaîne à tester avec le modèle [^\s*\d+(\,\d+)*\s*$] et fin pour arrêter :
  145,678
J'ai trouvé la correspondance [  145,678   ]
```

3.3 Intercepter les erreurs d'exécution

Une autre méthode de gestion des erreurs d'exécution est de les laisser se produire, d'en être avertis et de les gérer alors. Normalement lorsqu'une erreur se passe à l'exécution, WSH affiche un message d'erreur et le programme est arrêté. Deux instructions nous permettent de modifier ce fonctionnement :

1. **on error resume next**
   Cette instruction indique au système (WSH) que nous allons gérer les erreurs nous-mêmes. Après cette instruction, toute erreur est simplement ignorée. par le système.
2. **on error goto 0**
   Cette instruction nous ramène au fonctionnement normal de gestion des erreurs.

Lorsque l'instruction **on error resume next** est active, nous devons gérer nous-mêmes les erreurs qui peuvent survenir. L'objet **Err** nous y aide. Cet objet a diverses propriétés et méthodes dont nous retiendrons les deux suivantes :

- **number** : un nombre entier numéro de la dernière erreur qui s'est produite. 0 veut dire "pas d'erreur"
- **description** : le message d'erreur qu'aurait affiché le système si on n'avait pas émis l'instruction **on error resume next**

Regardons l'exemple qui suit :

Programme	Résultats
' erreur non gérée Option Explicit Dim nombre nombre=cdbl("abcd")	C:\ err5.vbs(6, 1) Erreur d'exécution Microsoft VBScript: Type incompatible: 'cdbl'

```
wscript.echo "nombre=" & nombre
```

Gérons maintenant l'erreur :

Programme	Résultats

```
' erreur gérée

Option Explicit
Dim nombre

' on gère les erreurs nous-mêmes
On Error Resume Next
nombre=cdbl("abcd")
' y-a-t-il eu erreur ?
If Err.number<>0 Then
  wscript.echo "L'erreur [" &
err.description & "] s'est produite"
  On Error GoTo 0
  wscript.quit 1
End If
' pas d'erreur - on revient au
fonctionnement normal
On Error GoTo 0
wscript.echo "nombre=" & nombre
wscript.quit 0
```

```
L'erreur [Type incompatible] s'est produite
```

Réécrivons le programme de saisie d'un entier >0 avec cette nouvelle méthode :

Programme

```
' lecture d'une donnée jusqu'à ce que celle-ci soit reconnue comme un nombre

Option Explicit

Dim fini, nombre

' on boucle tant que la donnée saisie n'est pas correcte
' la boucle est contrôlée par un booléen fini, mis à faux au départ (= ce n'est pas
fini)

fini=false
Do While Not fini
  ' on demande le nombre
  wscript.stdout.write "Tapez un nombre entier >0: "
  ' on le lit
  nombre=wscript.stdin.readLine
  ' on teste le format de la donnée lue
  On Error Resume Next
  nombre=cdbl(nombre)
  If err.number=0 Then
    ' pas d'erreur c'est un nombre
    ' on revient au mode normal de gestion des erreurs
    On Error GoTo 0
    ' est-ce un entier >0
    If (nombre-int(nombre))=0 And nombre>0 Then
      fini=true
    End If
  End If
  ' on revient au mode normal de gestion des erreurs
  On Error GoTo 0
  ' msg d'erreur éventuel
  If Not fini Then wscript.echo "Erreur, vous n'avez pas tapé un nombre entier >0.
Recommencez svp..."
Loop

' confirmation
wscript.echo "Merci pour le nombre entier >0 : " & nombre

' et fin
wscript.quit 0
```

Résultats

```
Tapez un nombre entier >0: 4.5
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: 4,5
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: abcd
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
Tapez un nombre entier >0: -4
Erreur, vous n'avez pas tapé un nombre entier >0. Recommencez svp...
```

```
Tapez un nombre entier >0: 1
Merci pour le nombre entier >0 : 1
```

Commentaires :

- Cette méthode est parfois la seule utilisable. Il ne faut alors pas oublier de revenir au mode normal de gestion des erreurs dès que la séquence d'instructions susceptible de générer l'erreur est terminée.

3.4 Application au programme de calcul d'impôts

Nous reprenons le programme de calcul d'impôts déjà écrit pour, cette fois, vérifier la validité des arguments passés au programme :

Programme

```
' calcul de l'impôt d'un contribuable
' le programme doit être appelé avec trois paramètres : marié enfants salaire
' marié : caractère O si marié, N si non marié
' enfants : nombre d'enfants
' salaire : salaire annuel sans les centimes

' aucune vérification de la validité des données n'est faite mais on
' vérifie qu'il y en a bien trois

' déclaration obligatoire des variables
Option Explicit
Dim syntaxe
syntaxe= _
    "Syntaxe : pg marié enfants salaire" & vbCRLF & _
    "marié : caractère O si marié, N si non marié" & vbCRLF & _
    "enfants : nombre d'enfants (entier >=0)" & vbCRLF & _
    "salaire : salaire annuel sans les centimes (entier >=0)"

' on vérifie qu'il y a 3 arguments
  Dim nbArguments
  nbArguments=wscript.arguments.count
  If nbArguments<>3 Then
    ' msg d'erreur
    wscript.echo syntaxe & vbCRLF & vbCRLF & "erreur : nombre d'arguments incorrect"
    ' arrêt avec code d'erreur 1
    wscript.quit 1
  End If

' on récupère les arguments en vérifiant leur validité
' un argument est transmis au programme sans espaces devant et derrière
' on utilisera des expression régulières pour vérifier la validité des données
  Dim modele, correspondances
  Set modele=new regexp

  ' le statut marital doit être parmi les caractères oOnN
  modele.pattern="^[oOnN]$"
  Set correspondances=modele.execute(wscript.arguments(0))
  If correspondances.count=0 Then
    ' erreur
    wscript.echo syntaxe & vbCRLF & vbCRLF & "erreur : argument marie incorrect"
    ' on quitte
    wscript.quit 2
  End If
  ' on récupère la valeur
  Dim marie
  If lcase(wscript.arguments(0)) = "o"Then
    marie=true
  Else
    marie=false
  End If

  ' enfants doit être un nombre entier >=0
  modele.pattern="^\d{1,2}$"
  Set correspondances=modele.execute(wscript.arguments(1))
  If correspondances.count=0 Then
    ' erreur
    wscript.echo syntaxe & vbCRLF & vbCRLF & "erreur : argument enfants incorrect"
    ' on quitte
    wscript.quit 3
  End If
  ' on récupère la valeur
  Dim enfants
  enfants=cint(wscript.arguments(1))
```

```
' salaire doit être un entier >=0
modele.pattern="^\d{1,9}$"
Set correspondances=modele.execute(wscript.arguments(2))
If correspondances.count=0 Then
  ' erreur
  wscript.echo syntaxe & vbCRLF & vbCRLF & "erreur : argument salaire incorrect"
  ' on quitte
  wscript.quit 4
End If
' on récupère la valeur
Dim salaire
salaire=clng(wscript.arguments(2))

' on définit les données nécessaire au calcul de l'impôt dans 3 tableaux
Dim limites, coeffn, coeffr
limites=array(12620,13190,15640,24740,31810,39970,48360, _
  55790,92970,127860,151250,172040,195000,0)
coeffr=array(0,0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45, _
  0.5,0.55,0.6,0.65)
coeffn=array(0,631,1290.5,2072.5,3309.5,4900,6898.5,9316.5, _
  12106,16754.5,23147.5,30710,39312,49062)

' on calcule le nombre de parts
Dim nbParts
If marie=true Then
  nbParts=(enfants/2)+2
Else
  nbParts=(enfants/2)+1
End If
If enfants>=3 Then nbParts=nbParts+0.5

' on calcule le quotient familial et le revenu imposable
Dim revenu, qf
revenu=0.72*salaire
qf=revenu/nbParts

' on calcule l'impôt
Dim i, impot
i=0
Do While i<ubound(limites) And qf>limites(i)
  i=i+1
Loop
impot=int(revenu*coeffr(i)-nbParts*coeffn(i))

' on affiche le résultat
wscript.echo "impôt=" & impot

' on quitte sans erreur
wscript.quit 0
```

Résultats

```
C:\>cscript impots2.vbs

Syntaxe : pg marié enfants salaire
marié : caractère O si marié, N si non marié
enfants : nombre d'enfants (entier >=0)
salaire : salaire annuel sans les centimes (entier >=0)

erreur : nombre d'arguments incorrect

C:\>cscript impots2.vbs a b c

Syntaxe : pg marié enfants salaire
marié : caractère O si marié, N si non marié
enfants : nombre d'enfants (entier >=0)
salaire : salaire annuel sans les centimes (entier >=0)

erreur : argument marie incorrect


C:\>cscript impots2.vbs o b c

Syntaxe : pg marié enfants salaire
marié : caractère O si marié, N si non marié
enfants : nombre d'enfants (entier >=0)
salaire : salaire annuel sans les centimes (entier >=0)

erreur : argument enfants incorrect

C:\>cscript impots2.vbs o 2 c
```

```
Syntaxe : pg marié enfants salaire
marié : caractère O si marié, N si non marié
enfants : nombre d'enfants (entier >=0)
salaire : salaire annuel sans les centimes (entier >=0)

erreur : argument salaire incorrect

C:\>cscript impots2.vbs o 2 200000

impôt=22504
```

4 Les fonctions et procédures

4.1 Les fonctions prédéfinies de vbscript

La richesse d'un langage dérive en grande partie de sa bibliothèque de fonctions, ces dernières pouvant être encapsulées dans des objets sous le nom de méthodes. Sous cet aspect, on peut considérer que vbscript est plutôt pauvre.
Le tableau suivant définit les fonctions de VBScript hors objets. Nous ne les détaillerons pas. Leur nom est en général une indication de leur rôle. Le lecteur consultera la documentation pour avoir des détails sur une fonction particulière.

Abs	Array	Asc	Atn
CBool	CByte	CCur	CDate
CDbl	Chr	CInt	CLng
Conversions	Cos	CreateObject	CSng
Date	DateAdd	DateDiff	DatePart
DateSerial	DateValue	Day	Derived Maths
Eval	Exp	Filter	FormatCurrency
FormatDateTime	FormatNumber	FormatPercent	GetLocale
GetObject	GetRef	Hex	Hour
InputBox	InStr	InStrRev	Int, Fixs
IsArray	IsDate	IsEmpty	IsNull
IsNumeric	IsObject	Join	LBound
LCase	Left	Len	LoadPicture
Log	LTrim; RTrim; and Trims	Maths	Mid
Minute	Month	MonthName	MsgBox
Now	Oct	Replace	RGB
Right	Rnd	Round	ScriptEngine
ScriptEngineBuildVersion	ScriptEngineMajorVersion	ScriptEngineMinorVersion	Second
SetLocale	Sgn	Sin	Space
Split	Sqr	StrComp	String
Tan	Time	Timer	TimeSerial
TimeValue	TypeName	UBound	UCase
VarType	Weekday	WeekdayName	Year

4.2 Programmation modulaire

Décrire la solution programmée d'un problème, c'est décrire la suite d'actions élémentaires exécutables par l'ordinateur et capables de résoudre le problème. Selon les langages ces opérations élémentaires sont plus ou moins sophistiquées. On trouve par exemple:

. lire une donnée provenant du clavier ou du disque
. écrire une donnée à l'écran, sur imprimante, sur disque, etc
. calculer des expressions
. se déplacer dans un fichier
. ...

Décrire un problème complexe peut nécessiter plusieurs milliers de ces instructions élémentaires et plus. Il est alors très difficile pour l'esprit humain d'avoir une vue globale d'un programme. Devant cette difficulté d'appréhender le problème dans sa globalité, on le décompose alors en sous-problèmes plus simples à résoudre. Considérons le problème suivant : Trier une liste de valeurs numériques tapées au clavier et afficher la liste triée à l'écran.

On peut dans un premier temps décrire la solution sous la forme suivante:

```
début
    lire les valeurs et les mettre dans un tableau T
    trier le tableau T
    écrire les valeurs triées du tableau T à l'écran
fin
```

On a décomposé le problème en 3 sous-problèmes, plus simples à résoudre. L'écriture algorithmique est souvent plus formalisée que la précédente et l'algorithme s'écrira plutôt:

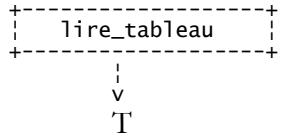
```
début
    lire_tableau(T)
    trier_tableau(T)
    écrire_tableau(T)
fin
```

où T représente un tableau. Les opérations

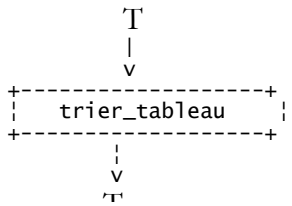
. lire_tableau(T)
. trier_tableau(T)
. écrire_tableau(T)

sont des opérations non élémentaires qui doivent être décrites à leur tour par des opérations élémentaires. Ceci est fait dans ce qu'on appelle des modules. La donnée T est appelée un paramètre du module. C'est une information que le programme appelant passe au module appelé (paramètre d'entrée) ou reçoit du module appelé (paramètre de sortie). Les paramètres d'un module sont donc les informations qui sont échangées entre le programme appelant et le module appelé.

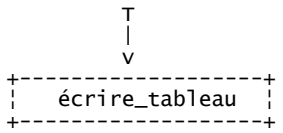
module *lire_tableau(T)*



module *trier_tableau(T)*



module *écrire_tableau(T)*



Le module *lire_tableau(T)* pourrait être décrit comme suit :

```
début
       écrire "Tapez la suite de valeurs à trier sous la forme val1 val2 ... : "
       lire valeurs
       construire tableau T à partir de la chaîne valeurs
fin
```

Ici, nous avons suffisamment décrit le module *lire_tableau*. En effet, les trois actions nécessaires ont une traduction immédiate en vbscript. La dernière nécessitera l'utilisation de la fonction split. Si vbscript n'avait pas cette fonction, l'action 3 devrait être décomposée à son tour en actions élémentaires ayant un équivalent immédiat en vbscript.

Le module écrire*_tableau(T)* pourrait être décrit comme suit :

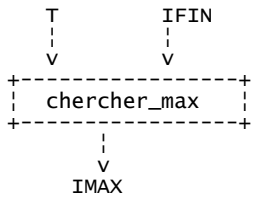
```
début
       construire chaîne texte "valeur1,valeur2,...." à partir du tableau T
       écrire texte
fin
```

Le module écrire*_tableau(T)* pourrait être décrit comme suit (on suppose que les indices des éléments de T commencent à 0) :

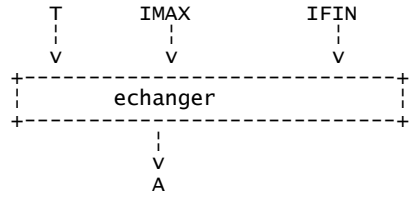
```
début
      N<-- indice dernier élément du tableau T
      pour IFIN variant de N à 1
      faire
          //on recherche l'indice IMAX du plus gd élément de T
          // IFIN est l'indice du dernier élément de T

          chercher_max(T, IFIN, IMAX)

          // on échange l'élément le plus grand de T avec le dernier élément de T

          échanger (T, IMAX, IFIN)

      finfaire
FIN
```

Ici l'algorithme utilise de nouveau des actions non élémentaires:
. *chercher_max(T, IFIN, IMAX)*
. *échanger(T, IMAX, IFIN)*

chercher_max(T, IFIN, IMAX) rend l'indice IMAX de l'élément le plus grand du tableau T dont l'indice du dernier élément est IFIN.



échanger(T, IMAX, IFIN) échange 2 éléments du tableau T , ceux d'indice IMAX et IFIN.



Il faut donc décrire les nouvelles opérations non élémentaires.

module *chercher_max(A, IFIN, IMAX)*

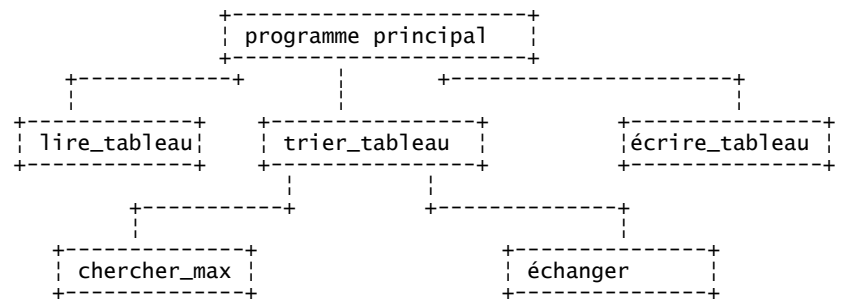
```
début
     IMAX<--0

     pour i variant de 1 à IFIN
     faire
        si T[i]>T[IMAX] alors
          début
             IMAX<--i
          fin
     finfaire
 fin
```

module *échanger(T IMAX, IFIN)*

```
début
     temp<----T[IMAX]
     T[IMAX]<---T[IFIN]
     T[IFIN]<---temp
fin
```

Le problème initial a été complètement décrit à l'aide d'opérations élémentaires vbscript et peut donc maintenant faire l'objet d'une traduction dans ce langage. On notera que les actions élémentaires peuvent différer d'un langage à l'autre et que donc l'analyse d'un problème doit à un certain moment tenir compte du langage de programmation utilisé. Un objet qui existe dans un langage peut ne pas exister dans un autre et modifier alors l'algorithme utilisé. Ainsi, si un langage avait une fonction de tri, il serait ici absurde de ne pas l'utiliser.

Le principe appliqué ici, est celui dit de l'analyse descendante. Si on représente l'ossature de la solution, on a la chose suivante :



On a une structure en arbre.

4.3 Les fonctions et procédures vbscript

Une fois l'analyse modulaire opérée, le programmeur peut traduire les modules de son algorithme en *fonctions* ou *procédures* vbscript. Les *fonctions* et *procédures* admettent toutes deux des paramètres d'entrée/sortie mais la *fonction* rend un résultat qui permet son utilisation dans des expressions alors que la *procédure* n'en rend pas.

4.3.1 Déclaration des fonctions et procédures vbscript

La déclaration d'une procédure vbscript est la suivante

```
sub nomProcédure([Byref/Byval] param1, [Byref/Byval] param2, ...)
      instructions
end sub
```

et celle d'une fonction

```
function nomFonction([Byref/Byval] param1, [Byref/Byval] param2, ...)
      instructions
end sub
```

Pour rendre son résultat, la fonction doit comporter une instruction d'affectation du résultat à une variable portant le nom de la fonction :

```
      nomFonction=résultat
```

L'exécution d'une fonction ou procédure s'arrête de deux façons :

1. à la rencontre de l'instruction de fin de fonction (`end function`) ou fin de procédure (`end sub`)
2. à la rencontre de l'instruction de sortie de fonction (`exit function`) ou de procédure (`exit sub`)

Pour la fonction, on se rappellera que le résultat doit avoir été affecté à une variable portant le nom de la fonction avant que celle-ci ne se termine par un `end function` ou `exit function`.

4.3.2 Modes de passage des paramètres d'une fonction ou procédure

Dans la déclaration des paramètres d'entrée-sortie d'une fonction ou procédure, on précise le mode (**byRef,byVal**) de transmission du paramètre du programme appelant vers le programme appelé :

```
sub nomProcédure([Byref/Byval] param1, [Byref/Byval] param2, ...)
function nomFonction([Byref/Byval] param1, [Byref/Byval] param2, ...)
```

Lorsque le mode de transmission *byRef* ou *byVal* n'est pas précisé, c'est le mode *byRef* qui est utilisé.

Paramètres effectifs, paramètres formels

Soit une fonction vbscript définie par

```
function nomFonction([Byref/Byval] paramForm1, [Byref/Byval] paramForm2, ...)
...
end function
```

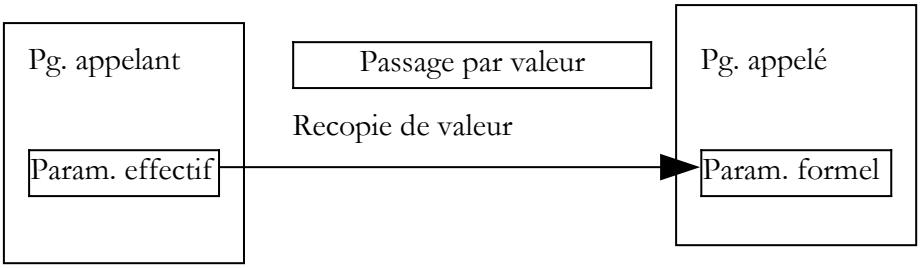
Les paramètres *parmamFormi* utilisés dans la définition de la fonction ou de la procédure sont appelés **paramètres formels**. La fonction précédente pourra être utilisée à partir du programme principal ou d'un autre module par une instruction du genre :

```
résultat=nomFonction(paramEff1, paramEff2, ...)
```

Les paramètres *parmamEffi* utilisés dans l'appel à la fonction ou la procédure sont appelés **paramètres effectifs**. Lorsque l'exécution de la fonction *nomFonction* commence, les paramètres formels reçoivent les valeurs des paramètres effectifs correspondants. Les mots clés *byRef* et *byVal* fixent le mode de transmission de ces valeurs.

Mode de transmission par valeur (byVal)

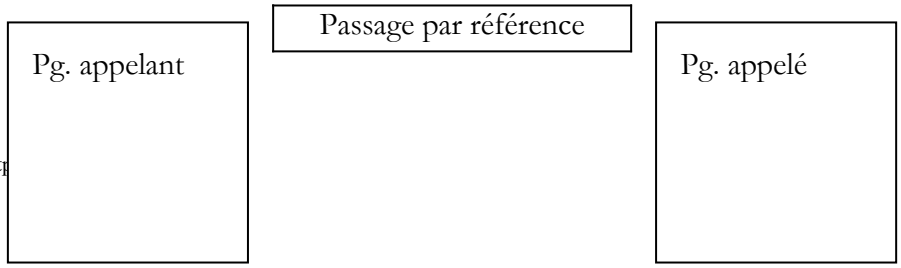
Lorsqu'un paramètre formel précise ce mode de transmission, le paramètre formel et le paramètre effectif sont alors deux variables différentes. La valeur du paramètre effectif est copiée dans le paramètre formel avant exécution de la fonction ou procédure. Si celle-ci modifie la valeur du paramètre formel au cours de son exécution, cela ne modifie en rien la valeur du paramètre effectif correspondant. Ce mode de transmission convient bien aux paramètres d'entrée de la fonction ou procédure.

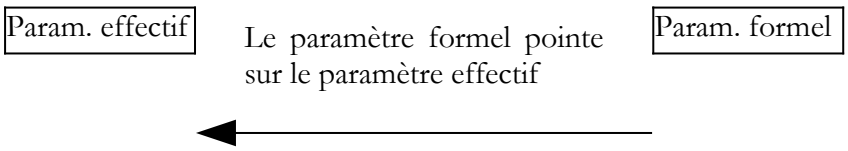


Mode de transmission par référence (byRef)

Ce mode de transmission est le mode par défaut si aucun mode de transmission du paramètre n'est indiqué. Lorsqu'un paramètre formel précise ce mode de transmission, le paramètre formel et le paramètre effectif correspondant **sont une seule et même variable**. Ainsi si la fonction modifie le paramètre formel, le paramètre effectif est également modifié. Ce mode de transmission convient bien :

- aux paramètres de sortie car la valeur de ceux-ci doivt être transmise au programme appelant
- aux paramètres d'entrée coûteux à recopier tels les tableaux





Le programme suivant montre des exemples de passage de paramètres :

Programme

```
Sub proc1(byval i, ByRef j, k)
  ' i est passé par valeur (byval) - le paramètre effectif et le paramètre formel sont
alors différents
  ' j est passé par valeur (byref) - le paramètre effectif et le paramètre formel sont
alors identiques
  ' le mode de passage de k n'est pas précisé. Par défaut, c'est par référence
  i=i+1
  j=j+1
  k=k+1
  affiche "dans proc1",i,j,k
End Sub

 Sub affiche(byval msg, ByVal i, ByVal j, ByVal k)
  ' affiche les valeurs de i et j et k
  wscript.echo msg & " i=" & i & " j=" & j & " k=" & k
End Sub


' ------------- appels aux fonctions et procédures

' init i et j
  i=4:j=5 : k=6

' vérification
  affiche "dans programme principal, avant l'appel à proc1 :",i,j,k

' appel procédure proc1
  proc1 i,j,k

' vérification
  affiche "dans programme principal, après l'appel à proc1 :",i,j,k

' fin
  wscript.quit 0
```

Résultats

```
dans programme principal, avant l'appel à proc1 : i=4 j=5 k=6
dans proc1 i=5 j=6 k=7
dans programme principal, après l'appel à proc1 : i=4 j=6 k=7
```

Commentaires

- Dans un script vbscript, il n'y a pas de place particulière pour les fonctions et les procédures. Elles peuvent être n'importe où dans le texte source. En général, on les regroupe soit au début soit à la fin et on fait en sorte que le programme principal constitue un bloc continu.

4.3.3 Syntaxe d'appel des fonctions et procédures

Soit une procédure p admettant des paramètres formels pf1, pf2, ...

- l'appel à la procédure p se fait sous la forme
  ```
  p pe1, pe2, ...
  ```
  sans parenthèses autour des paramètres

- si la procédure p n'admet aucun paramètre, on peut indifféremment utiliser l'appel p ou p() et la déclaration *sub p* ou sub p*()*

Soit une fonction f admettant des paramètres formels pf1, pf2, ...

- l'appel à la fonction f se fait sous la forme
  `résultat=f(pe1, pe2, ...)`
  les parenthèses autour des paramètres sont obligatoires. Si la fonction f n'admet aucun paramètre, on peut indifféremment utiliser l'appel *f* ou *f()* et la déclaration *function f* ou *function f()*.
- le résultat de la fonction f peut être ignoré par le programme appelant. La fonction f est alors considérée comme une procédure et suit les règles d'appel des procédures. On écrit alors `f pe1, pe2, ...` (sans parenthèses) pour appeler la fonction f.

Si la fonction ou procédure est une méthode d'objet, il semblerait que les règles soient quelque peu différentes et non homogènes.

- ainsi on peut écrire `MyFile.WriteLine "Ceci est un test."` ou `MyFile.WriteLine("Ceci est un test.")`
- mais si on peut écrire `wscript.echo 4`, on ne peut pas écrire `wscript.echo(4)`.

On s'en tiendra aux règles suivantes :

- pas de parenthèses autour des paramètres d'une procédure ou d'une fonction utilisée comme une procédure
- parenthèses autour des paramètres d'une fonction

4.3.4 Quelques exemples de fonctions

On trouvera ci-dessous quelques exemples de définitions et utilisations de fonctions :

Programme

```
 Function plusgrandque(byval i, ByVal j)
  ' rend le booléen vrai si i>j, le booléen faux sinon

  ' vérification des données
  If isnumeric(i) And isnumeric(j) Then
    If i>j Then
      plusgrandque=true
    Else
      plusgrandque=false
    End If
  Else
    wscript.echo "Arguments (" & i & "," & j & ") erronés"
    plusgrandque=false
  End If
  Exit Function
End Function

 Function rendUnTableau(byval n)
  ' rend un tableau de n éléments
  tableau=array()
  ' vérification validité du paramètre n
  If isnumeric(n) And n>=1 Then
    ReDim Preserve tableau(n)
    For i= 0 To n-1
      tableau(i)=i
    Next
  Else
    wscript.echo "Argument [" & n & "] erroné"
  End If
  ' on rend le résultat
  rendUnTableau=tableau
End Function

 Function argumentsVariables(byref arguments)
  ' arguments est un tableau de nombres dont on rend la somme
  somme=0
```

```
  For i=0 To ubound(arguments)
    somme=somme+arguments(i)
  Next
  argumentsVariables=somme
End Function

  ' deux fonctions sans paramètres déclarées de 2 façons différentes
  Function sansParametres1
    sansParametres=4
  End Function

  Function sansParametres2()
    sansParametres=4
  End Function


' ------------- appels aux fonctions et procédures

' appels fonction plusgrandque
  wscript.echo "plusgrandque(10,6)=" & plusgrandque(10,6)
  wscript.echo "plusgrandque(6,10)=" & plusgrandque(6,10)
  wscript.echo "plusgrandque(6,6)=" & plusgrandque(6,6)
  wscript.echo "plusgrandque(6,'a')=" & plusgrandque(6,"a")

' appels à la fonction rendUnTableau
  monTableau=rendUnTableau(10)
  For i=0 To ubound(monTableau)
    wscript.echo monTableau(i)
  Next
  monTableau=rendUnTableau(-6)
  For i=0 To ubound(monTableau)
    wscript.echo monTableau(i)
  Next

' appels à la fonction argumentsVariables
  wscript.echo "somme=" & argumentsVariables(array(-1,2,7,8))
  wscript.echo "somme=" & argumentsVariables(array(-1,10,12))

' appels des fonctions sans paramètres
  res=sansParametres1
  res=sansParametres1()
  sansParametres1
  sansParametres1()

  res=sansParametres2
  res=sansParametres2()
  sansParametres2
  sansParametres2()

' fin
  wscript.quit 0
```

Résultats

```
plusgrandque(10,6)=Vrai
plusgrandque(6,10)=Faux
plusgrandque(6,6)=Faux
Arguments (6,a) erronés
plusgrandque(6,'a')=Faux
0
1
2
3
4
5
6
7
8
9

Argument [-6] erroné
somme=16
somme=21
somme=10
```

Commentaires

- la fonction *rendUnTableau* montre qu'une fonction peut rendre plusieurs résultats et non un seul. Il suffit qu'elle les place dans un variant tableau et qu'elle rende ce variant comme résultat.

- inversement la fonction *argumentsVariables* montre qu'on peut écrire une fonction qui admet un nombre variable d'arguments. Il suffit là également de les mettre dans un variant tableau et de faire de ce variant un paramètre de la fonction.

4.3.5 Paramètre de sortie ou résultat d'une fonction

Supposons que l'analyse d'une application ait montré la nécessité d'un module M avec des paramètres d'entrée Ei et des paramètres de sortie Sj. Rappelons que les paramètres d'entrée sont des informations que le programme appelant donne au programme appelé et qu'inversement les paramètres de sortie sont des informations que le programme appelé donne au programme appelant. On a en vbscript plusieurs solutions pour les paramètres de sortie :

- s'il n'y a qu'un seul paramètre de sortie, on peut en faire le résultat d'une fonction. Il n'y a alors plus de paramètre de sortie mais simplement un résultat de fonction.
- s'il y a n paramètres de sortie, l'un d'entre-eux peut servir de résultat de fonction , les n-1 autres restant des paramètres de sortie. On peut aussi ne pas utiliser de fonction mais une procédure à n paramètres de sortie. On peut également utiliser une fonction qui rendra un tableau dans lequel on aura placé les n valeurs à rendre au programme appelant. On se rappellera que le programme appelé rend ses résultats au programme appelant par recopie de valeurs. Cette recopie est évitée dans le cas de paramètres de sortie passés par référence. Il y a donc dans cette dernière solution un gain de temps.

4.4 Le programme Vbscript de tri de valeurs

Nous avions commencé la discussion sur la programmation modulaire par l'étude algorithmique d'un tri de valeurs numériques tapées au clavier. Voici la traduction vbscript qui pourrait en être faite :

Programme

```
' programme principal
Option Explicit

Dim T        ' le tableau de valeurs à trier

' lecture des valeurs
T=lire_tableau
' tri des valeurs
trier_tableau T
' affichage des valeurs triées
ecrire_tableau T

' fin
wscript.quit 0

' ---------- fonctions & procédures

' -------- lire_tableau
Function lire_tableau
  ' on demande les valeurs
  wscript.stdout.write "Tapez les valeurs à trier sous la forme val1 val2  ... valn : "
  ' on les lit
  Dim valeurs
  valeurs=wscript.stdin.readLine
  ' on les met dans un tableau
  lire_tableau=split(valeurs," ")
End Function

' -------- ecrire_tableau
Sub ecrire_tableau(byref T)
  ' affiche le contenu du tableau T
  wscript.echo join(T," ")
End Sub

' -------- trier_tableau
Sub trier_tableau (byref T)
  ' tri le tableau T en ordre croissant

  ' on cherche l'indice imax du tableau T[0..ifin]
  ' pour échanger T[imax] avec le dernier élément du tableau T[0..ifin]
  ' ensuite on recommence avec un tableau ayant 1 élément de moins
```

```
  Dim ifin, imax, temp
  For ifin=ubound(T) To 1 Step -1
    ' on cherche l'indice imax du tableau T[0..ifin]
    imax=chercher_max(T,ifin)
    ' on l'échange le max avec le dernier élément du tableau T[0..ifin]
    temp=T(ifin):T(ifin)=T(imax):T(imax)=temp
  Next
End Sub

' -------- chercher_max
Function chercher_max(byRef T, ByVal ifin)
  ' on cherche l'indice imax du tableau T[0..ifin]
  Dim i, imax
  imax=0
  For i=1 To ifin
    If cdbl(T(i))>cdbl(T(imax)) Then imax=i
  Next

  ' On rend le résultat
  chercher_max=imax
End Function
```

Résultats

```
Tapez les valeurs à trier sous la forme val1 val2  ... valn : 10 9 8 7 6 1
1 6 7 8 9 10
```

Commentaires :

- le module *échanger* qui avait été identifié dans l'algorithme initial n'a pas fait ici l'objet d'un module en vbscript parce que jugé trop simple pour faire l'objet d'un module particulier.

4.5 Le programme IMPOTS sous forme modulaire

Nous reprenons le programme de calcul de l'impôt écrit cette fois sous forme modulaire

Programme

```
' calcul de l'impôt d'un contribuable
' le programme doit être appelé avec trois paramètres : marié enfants salaire
' marié : caractère O si marié, N si non marié
' enfants : nombre d'enfants
' salaire : salaire annuel sans les centimes

' déclaration obligatoire des variables
  Option Explicit
  Dim erreur

' on récupère les arguments en vérifiant leur validité
  Dim marie, enfants, salaire
  erreur=getArguments(marie,enfants,salaire)
  ' erreur ?
  If erreur(0)<>0 Then wscript.echo erreur(1) : wscript.quit erreur(0)

' on récupère les données nécessaires au calcul de l'impôt
  Dim limites, coeffR, coeffN
  getData limites,coeffR,coeffN

  ' on affiche le résultat
  wscript.echo "impôt=" & calculerImpot(marie,enfants,salaire,limites,coeffR,coeffN)

  ' on quitte sans erreur
  wscript.quit 0

' ------------ fonctions et procédures

' ----------- getArguments
Function getArguments(byref marie, ByRef enfants, ByRef salaire)
  ' doit récupérer trois valeurs passées comme argument au programme principal
  ' un argument est transmis au programme sans espaces devant et derrière
  ' on utilisera des expression régulières pour vérifier la validité des données

  ' rend un variant tableau erreur à 2 valeurs
  ' erreur(0) : code d'erreur, 0 si pas d'erreur
  ' erreur(1) : message d'erreur si erreur sinon la chaîne vide

  Dim syntaxe
  syntaxe= _
    "Syntaxe : pg marié enfants salaire" & vbCRLF & _
```

```
    "marié : caractère O si marié, N si non marié" & vbCRLF & _
    "enfants : nombre d'enfants (entier >=0)" & vbCRLF & _
    "salaire : salaire annuel sans les centimes (entier >=0)"

' on vérifie qu'il y a 3 arguments
  Dim nbArguments
  nbArguments=wscript.arguments.count
  If nbArguments<>3 Then
    ' msg d'erreur
    getArguments= array(1,syntaxe & vbCRLF & vbCRLF & "erreur : nombre d'arguments
incorrect")
    ' fin
    Exit Function
  End If

  Dim modele, correspondances
  Set modele=new regexp

  ' le statut marital doit être parmi les caractères oOnN
  modele.pattern="^[oOnN]$"
  Set correspondances=modele.execute(wscript.arguments(0))
  If correspondances.count=0 Then
    ' msg d'erreur
    getArguments=array(2,syntaxe & vbCRLF & vbCRLF & "erreur : argument marie
incorrect")
    ' on quitte
    Exit Function
  End If
  ' on récupère la valeur
  If lcase(wscript.arguments(0)) = "o"Then
    marie=true
  Else
    marie=false
  End If

  ' enfants doit être un nombre entier >=0
  modele.pattern="^\d{1,2}$"
  Set correspondances=modele.execute(wscript.arguments(1))
  If correspondances.count=0 Then
    ' erreur
    getArguments= array(3,syntaxe & vbCRLF & vbCRLF & "erreur : argument enfants
incorrect")
    ' on quitte
    Exit Function
  End If
  ' on récupère la valeur
  enfants=cint(wscript.arguments(1))

  ' salaire doit être un entier >=0
  modele.pattern="^\d{1,9}$"
  Set correspondances=modele.execute(wscript.arguments(2))
  If correspondances.count=0 Then
    ' erreur
    getArguments= array(4,syntaxe & vbCRLF & vbCRLF & "erreur : argument salaire
incorrect")
    ' on quitte
    Exit Function
  End If
  ' on récupère la valeur
  salaire=clng(wscript.arguments(2))

  ' c'est fini sans erreur
  getArguments=array(0,"")
End Function

' ----------- getData
Sub getData(byref limites, ByRef coeffR, ByRef coeffN)
  ' on définit les données nécessaire au calcul de l'impôt dans 3 tableaux
  limites=array(12620,13190,15640,24740,31810,39970,48360, _
    55790,92970,127860,151250,172040,195000,0)
  coeffr=array(0,0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45, _
    0.5,0.55,0.6,0.65)
  coeffn=array(0,631,1290.5,2072.5,3309.5,4900,6898.5,9316.5, _
    12106,16754.5,23147.5,30710,39312,49062)
End Sub

' ----------- calculerImpot
Function calculerImpot(byval marie,ByVal enfants,ByVal salaire, ByRef limites, ByRef
coeffR, ByRef coeffN)

  ' on calcule le nombre de parts
  Dim nbParts
  If marie=true Then
    nbParts=(enfants/2)+2
  Else
    nbParts=(enfants/2)+1
```

```
  End If
  If enfants>=3 Then nbParts=nbParts+0.5

  ' on calcule le quotient familial et le revenu imposable
  Dim revenu, qf
  revenu=0.72*salaire
  qf=revenu/nbParts

  ' on calcule l'impôt
  Dim i, impot
  i=0
  Do While i<ubound(limites) And qf>limites(i)
    i=i+1
  Loop
  calculerImpot=int(revenu*coeffr(i)-nbParts*coeffn(i))
End Function
```

Commentaires

- la fonction **getArguments** permet de récupérer les informations (marie, enfants, salaire) du contribuable. Ici, elles sont passées en arguments au programme vbscript. Si cela devait changer, par exemple si ces arguments venaient d'une interface graphique, seule la procédure *getArguments* devrait être réécrite et pas les autres.
- la fonction *getArguments* peut détecter des erreurs sur les arguments. Lorsque ceci se produit, on aurait pu décider d'arrêter l'exécution du programme dans la fonction getArguments par une instruction *wscript.quit*. Ceci ne doit jamais être fait dans une fonction ou procédure. Si une fonction ou procédure détecte une erreur, elle doit le signaler d'une façon ou d'une autre au programme appelant. C'est à lui de prendre la décision d'arrêter l'exécution ou non, pas à la procédure. Dans notre exemple, le programme appelant pourrait décider de redemander à l'utilisateur de retaper la donnée erronée au clavier plutôt que d'arrêter l'exécution.
- ici, la fonction *getArguments* rend un variant tableau où le 1er élément est un code d'erreur (0 si pas d'erreur) et le second un message d'erreur si il y a eu erreur. En testant le résultat obtenu, le programme appelant peut savoir s'il y a eu erreur ou non.
- la procédure **getData** permet d'obtenir les données nécessaires au calcul de l'impôt. Ici elles sont directement définies dans la procédure getData. Si ces données devaient provenir d'une autre source, d'un fichier ou d'une base de données par exemple, seule la procédure getData devrait être réécrite et pas les autres.
- la fonction **calculerImpot** permet de calculer l'impôt une fois que toutes les données ont été obtenues quelque soit la façon dont elles ont été obtenues.
- on notera donc qu'une écriture modulaire permet une (ré)utilisation de certains modules dans différents contextes. Ce concept a été dans les vingt dernières années fortement développé dans le concept d'objet.

5 Les fichiers texte

Un fichier texte est un fichier contenant des lignes de texte. Examinons la création et l'utilisation de tels fichiers sur des exemples.

5.1 Création et utilisation

Programme

```vbscript
' création & remplissage d'un fichier texte
Option Explicit
Dim objFichier,MyFile
Const ForReading = 1, ForWriting = 2, ForAppending = 8

' on crée un objet fichier
Set objFichier=CreateObject("Scripting.FileSystemObject")
' on crée un fichier texte
Set MyFile= objFichier.OpenTextFile("testfile.txt", ForWriting, True)
' on écrit une ligne de texte dedans
MyFile.WriteLine "ligne1"
' on ferme le fichier texte
MyFile.Close

' on lui ajoute des éléments
Set MyFile= objFichier.OpenTextFile("testfile.txt", ForAppending, True)
' on écrit une ligne de texte dedans
MyFile.WriteLine "ligne2" & vbCRLF & "ligne3"
' on ferme le fichier texte
MyFile.Close

' on ouvre le fichier en lecture
Set MyFile= objFichier.OpenTextFile("testfile.txt", ForReading)
' on lit tout le contenu
dim ligne
Do While Not MyFile.AtEndOfStream
  ligne=MyFile.ReadLine
  wscript.echo ligne
Loop
' on ferme le fichier texte
MyFile.Close

' fin
wscript.quit 0
```

Résultats

```
C:\>cscript fic1.vbs

C:\>dir

 FIC1     VBS           352  07/01/02   7:07 fic1.vbs
TESTFILE TXT            25  07/01/02   7:07 testfile.txt

C:\>more testfile.txt

Ceci est un autre test.
```

Commentaires

- la ligne 7 crée un objet fichier de type "*Scripting.FileSystemObject*" par la fonction *CreateObject("Scripting.FileSystemObject")*. Un tel objet permet l'accès à tout fichier du système pas simplement à des fichiers texte.
- la ligne 9 crée un objet "*TextStream*". La création de cet objet est associée à la création du fichier *testfile.txt*. Ce fichier n'est pas désigné par un nom absolu du genre *c:\dir1\dir2\....\testfile.txt* mais par un nom relatif *testfile.txt*. Il sera alors créé dans le répertoire d'où sera lancée la commande d'exécution du fichier.
- le système de fichiers du système windows n'a pas connaissance de concepts tels que fichier texte ou fichier non texte. Il ne connaît que des fichiers. C'est donc au programme qui exploite ce fichier de savoir s'il va le traiter comme un fichier texte ou non.
- La ligne 9 crée un objet d'où la commande *set* utilisée pour l'affectation. La création d'un objet fichier texte passe par la création de 2 objets :
  - la création d'un objet *Scripting.FileSystemObject* (ligne 7)
  - puis par la création d'un objet "*TextStream*" (fichier texte) par la méthode *OpenTextFile* de l'objet *Scripting.FileSystemObject* qui admet plusieurs paramètres :
    - le nom du fichier à gérer (obligatoire)
    - le mode d'utilisation du fichier. C'est un entier avec 3 valeurs possibles :
      - 1 : utilisation du fichier en lecture
      - 2 : utilisation du fichier en écriture. S'il n'existe pas déjà et si le 3ième paramètre est présent et a la valeur true, il est créé sinon il n'est pas. S'il existe déjà, il est écrasé.

- 8 : utilisation du fichier en ajout, c.a.d. écriture en fin de fichier. Si le fichier n'existe pas déjà et si le 3ième paramètre est présent et a la valeur true, il est créé sinon il n'est pas.

- la ligne 11 écrit une ligne de texte avec la méthode *WriteLine* de l'objet TextStream créé.
- la ligne 13 "ferme" le fichier. On ne peut alors plus écrire ou lire dedans.
- la ligne 16 crée un nouvel objet "*TextStream*" pour exploiter le même fichier que précédemment mais cette fois-ci en mode "ajout". Les lignes qui seront écrites le seront derrière les lignes existantes.
- la ligne 18 écrit deux nouvelles lignes sachant que la constante vbCRLF est la marque de fin de ligne des fichiers texte.
- la ligne 20 ferme de nouveau le fichier
- la ligne 23 le rouvre en mode "lecture" : on va lire le contenu du fichier.
- La ligne 27 lit une ligne de texte avec la méthode *ReadLine* de l'objet *TextStream*. Lorsque le fichier vient d'être "ouvert", on est positionné sur la 1ère ligne de texte de celui-ci. Lorsque celle-ci a été lue par la méthode *ReadLine*, on est positionné sur la seconde ligne. Ainsi la méthode *Readline* non seulement lit la ligne courante mais "avance" ensuite automatiquement à la ligne suivante.
- Pour ligne toutes les lignes de texte, la méthode *ReadLine* doit être appliquée de façon répétée dans une boucle. Celle-ci (ligne 26) se termine lorsque l'attribut *AtEndOfStream* de l'objet *TextStream* a la valeur *true*. Cela signifie alors qu'il n'y a plus de lignes à lire dans le fichier.

5.2 Les cas d'erreur

On rencontre deux cas d'erreur fréquents :

- ouverture en lecture d'un fichier qui n'existe pas
- ouverture en écriture ou ajout d'un fichier qui n'existe pas avec comme le troisième paramètre à false dans l'appel à la méthode *OpenTextFile*.

Le programme suivant montre comment détecter ces erreurs :

Programme

```
' création & remplissage d'un fichier texte
Option Explicit
Dim objFichier,MyFile
Const ForReading = 1, ForWriting = 2, ForAppending = 8
Dim codeErreur

' on crée un objet fichier
Set objFichier=CreateObject("Scripting.FileSystemObject")

' on ouvre un fichier texte devant exister en lecture
On Error Resume Next
Set MyFile= objFichier.OpenTextFile("abcd", ForReading)
codeErreur=err.number
On Error GoTo 0
If codeErreur<>0 Then
  ' le fichier n'existe pas
  wscript.echo "Le fichier [abcd] n'existe pas"
Else
  ' on ferme le fichier texte
  MyFile.Close
End If


' on ouvre un fichier texte devant exister en écriture
On Error Resume Next
Set MyFile= objFichier.OpenTextFile("abcd", ForWriting, False)
codeErreur=err.number
On Error GoTo 0
If codeErreur<>0 Then
  wscript.echo "Le fichier [abcd] n'existe pas"
Else
  ' on ferme le fichier texte
```

```
  MyFile.Close
End If

' on ouvre un fichier texte devant exister en ajout
On Error Resume Next
Set MyFile= objFichier.OpenTextFile("abcd", ForAppending, False)
codeErreur=err.number
On Error GoTo 0
If codeErreur<>0 Then
  wscript.echo "Le fichier [abcd] n'existe pas"
Else
  ' on ferme le fichier texte
  MyFile.Close
End If

' fin
wscript.quit 0
```

Résultats

```
C:\>dir

FIC1     VBS           964  07/01/02   7:54 fic1.vbs
TESTFILE TXT             0  07/01/02   8:18 testfile.txt
FIC2     VBS         1 252  07/01/02   8:23 fic2.vbs
         3 fichier(s)              2 216 octets
         2 répertoire(s)        4 007.11 Mo libre

C:\>cscript fic2.vbs

Le fichier [abcd] n'existe pas
Le fichier [abcd] n'existe pas
Le fichier [abcd] n'existe pas
```

5.3 L'application IMPOTS avec un fichier texte

Nous reprenons l'application de calcul de l'impôt en supposant que les données nécessaires au calcul de l'impôt sont dans un fichier texte appelé *data.txt* :

```
12620 13190 15640 24740 31810 39970 48360 55790 92970 127860 151250 172040 195000 0
0 0,05 0,1 0,15 0,2 0,25 0,3 0,35 0,4 0,45 0,5 0,55 0,6 0,65
0 631 1290,5 2072,5 3309,5 4900 6898,5 9316,5 12106 16754,5 23147,5 30710 39312 49062
```

Les trois lignes contiennent respectivement les données des tableaux limites, coeffR et coeffN de l'application. Grâce à la modularisation de notre application, les modifications interviennent essentiellement dans la procédure **getData** chargée de construire les trois tableaux. Le nouveau programme est le suivant :

Programme

```
' calcul de l'impôt d'un contribuable
' le programme doit être appelé avec trois paramètres : marié enfants salaire
' marié : caractère O si marié, N si non marié
' enfants : nombre d'enfants
' salaire : salaire annuel sans les centimes

' déclaration obligatoire des variables
  Option Explicit
  Dim erreur

' on récupère les arguments en vérifiant leur validité
  Dim marie, enfants, salaire
  erreur=getArguments(marie,enfants,salaire)
  ' erreur ?
  If erreur(0)<>0 Then wscript.echo erreur(1) : wscript.quit erreur(0)

' on récupère les données nécessaires au calcul de l'impôt
  Dim limites, coeffR, coeffN
  erreur=getData(limites,coeffR,coeffN)
  ' erreur ?
  If erreur(0)<>0 Then wscript.echo erreur(1) : wscript.quit 5

  ' on affiche le résultat
  wscript.echo "impôt=" & calculerImpot(marie,enfants,salaire,limites,coeffR,coeffN)

  ' on quitte sans erreur
  wscript.quit 0
```

```
' ------------ fonctions et procédures

' ----------- getArguments
Function getArguments(byref marie, ByRef enfants, ByRef salaire)
  ' doit récupérer trois valeurs passées comme argument au programme principal
  ' un argument est transmis au programme sans espaces devant et derrière
  ' on utilisera des expression régulières pour vérifier la validité des données

  ' rend un variant tableau erreur à 2 valeurs
  ' erreur(0) : code d'erreur, 0 si pas d'erreur
  ' erreur(1) : message d'erreur si erreur sinon la chaîne vide

  Dim syntaxe
  syntaxe= _
    "Syntaxe : pg marié enfants salaire" & vbCRLF & _
    "marié : caractère O si marié, N si non marié" & vbCRLF & _
    "enfants : nombre d'enfants (entier >=0)" & vbCRLF & _
    "salaire : salaire annuel sans les centimes (entier >=0)"

' on vérifie qu'il y a 3 arguments
  Dim nbArguments
  nbArguments=wscript.arguments.count
  If nbArguments<>3 Then
    ' msg d'erreur
    getArguments= array(1,syntaxe & vbCRLF & vbCRLF & "erreur : nombre d'arguments
incorrect")
    ' fin
    Exit Function
  End If

  Dim modele, correspondances
  Set modele=new regexp

  ' le statut marital doit être parmi les caractères oOnN
  modele.pattern="^[oOnN]$"
  Set correspondances=modele.execute(wscript.arguments(0))
  If correspondances.count=0 Then
    ' msg d'erreur
    getArguments=array(2,syntaxe & vbCRLF & vbCRLF & "erreur : argument marie
incorrect")
    ' on quitte
    Exit Function
  End If
  ' on récupère la valeur
  If lcase(wscript.arguments(0)) = "o"Then
    marie=true
  Else
    marie=false
  End If

  ' enfants doit être un nombre entier >=0
  modele.pattern="^\d{1,2}$"
  Set correspondances=modele.execute(wscript.arguments(1))
  If correspondances.count=0 Then
    ' erreur
    getArguments= array(3,syntaxe & vbCRLF & vbCRLF & "erreur : argument enfants
incorrect")
    ' on quitte
    Exit Function
  End If
  ' on récupère la valeur
  enfants=cint(wscript.arguments(1))

  ' salaire doit être un entier >=0
  modele.pattern="^\d{1,9}$"
  Set correspondances=modele.execute(wscript.arguments(2))
  If correspondances.count=0 Then
    ' erreur
    getArguments= array(4,syntaxe & vbCRLF & vbCRLF & "erreur : argument salaire
incorrect")
    ' on quitte
    Exit Function
  End If
  ' on récupère la valeur
  salaire=clng(wscript.arguments(2))

  ' c'est fini sans erreur
  getArguments=array(0,"")
End Function

' ----------- getData
Function getData(byref limites, ByRef coeffR, ByRef coeffN)
  ' les données des trois tableaux limites, coeffR, coeffN sont dans un fichier texte
  ' appelé data.txt. Chaque tableau occupe une ligne sous la forme val1 val2 ... valn
  ' on trouve dans l'ordre limites, coeffR, coeffN
```

```
  ' rend un variant erreur tableau à 2 éléments pour gérer l'éventuelle erreur
  ' erreur(0) : 0 si pas d'erreur, un nombre entier >0 sinon
  ' erreur(1) : le message d'erreur si erreur

  Dim objFichier,MyFile,codeErreur
  Const ForReading = 1, dataFileName="data.txt"

  ' on crée un objet fichier
  Set objFichier=CreateObject("Scripting.FileSystemObject")
  ' on ouvre le fichier data.txt en lecture
  On Error Resume Next
  Set MyFile= objFichier.OpenTextFile(dataFileName, ForReading)
  ' erreur ?
  codeErreur=err.number
  On Error GoTo 0
  If codeErreur<>0 Then
    ' il y a eu erreur - on la note
    getData=array(1,"Impossible d'ouvrir le fichier des données [" & dataFileName & "]
en lecture")
    ' on rentre
    Exit Function
  End If

  ' on suppose maintenant que le contenu est correct et on ne fait aucune vérification
  ' on lit les 3 lignes

  ' limites
  Dim ligne, i
  ligne=MyFile.ReadLine
  getDataFromLine ligne,limites

  ' coeffR
  ligne=MyFile.ReadLine
  getDataFromLine ligne,coeffR


  ' coeffN
  ligne=MyFile.ReadLine
  coeffN=split(ligne," ")
  getDataFromLine ligne,coeffN

  ' on ferme le fichier
  MyFile.close

  ' c'est fini sans erreur
  getData=array(0,"")
End Function

' ----------- getDataFromLine
Sub getDataFromLine(byref ligne, ByRef tableau)
  ' met dans tableau les valeurs numériques contenues dans ligne
  ' celles-ci sont séparées par un ou plusieurs espaces

  ' au départ le tableau est vide
  tableau=array()
  ' on définit un modèle pour la ligne
  Dim modele, correspondances
  Set modele= New RegExP
  With modele
    .pattern="\d+,\d+|\d+"    ' 140,5 ou 140
    .global=true              ' toutes les valeurs
  End With

  ' on analyse la ligne
  Set correspondances=modele.execute(ligne)
  Dim i
  For i=0 To correspondances.count-1
    ' on redimensionne le tableau
    ReDim Preserve tableau(i)
    ' on affecte une valeur au nouvel élément
    tableau(i)=cdbl(correspondances(i).value)
  Next

  'fin
End Sub



' ----------- calculerImpot
Function calculerImpot(byval marie,ByVal enfants,ByVal salaire, ByRef limites, ByRef
coeffR, ByRef coeffN)

  ' on calcule le nombre de parts
  Dim nbParts
  If marie=true Then
```

```
    nbParts=(enfants/2)+2
  Else
    nbParts=(enfants/2)+1
  End If
  If enfants>=3 Then nbParts=nbParts+0.5

  ' on calcule le quotient familial et le revenu imposable
  Dim revenu, qf
  revenu=0.72*salaire
  qf=revenu/nbParts

  ' on calcule l'impôt
  Dim i, impot
  i=0
  Do While i<ubound(limites) And qf>limites(i)
    i=i+1
  Loop
  calculerImpot=int(revenu*coeffr(i)-nbParts*coeffn(i))
End Function
```

Commentaires :

- dans le fichier texte **data.txt**, les valeurs peuvent être séparées par un ou plusieurs espaces, d'où l'impossibilité d'utiliser la fonction split pour récupérer les valeurs de la ligne. Il a fallu passer par une expression régulière
- la fonction **getData** rend, outre les trois tableaux limites, coeffR, coeffN, un résultat indiquant s'il y a eu erreur ou non. Ce résultat est un variant tableau de eux éléments. Le premier élément est un code d'erreur (0 si pas d'erreur), le second le message d'erreur s'il y a eu erreur.
- la fonction *getData* ne teste pas la validité des valeurs trouvées dans le fichier data.txt. En situation réelle, elle devrait le faire.

Table of Contents