Apprendre la programmation en VBA pour EXCEL par la pratique - Première partie

Tome 1 - Des bases de la programmation à l'algorithme de classement rapide QuickRanking

Par Laurent OTT

Date de publication : 2 juillet 2016

Il s'agit d'une compilation d'exemples pratiques pour apprendre la programmation en Microsoft Office Excel avec VBA.
Vous allez aussi apprendre des exemples d'Excel avec les autres composants de Microsoft Office tels que Outlook et Access.
C'est un cours qui peut servir de support de formation pratique pour apprendre la programmation VBA sous Microsoft Office Excel. Les exemples sont en Excel 2010, mais peuvent être adaptés pour les versions récentes : Excel 2013 et Excel 2016.

Merci pour vos avis.
Commentez

En complément sur Developpez.com

- Mémento sur la programmation en VBA pour EXCEL - Tome 2
- Mémento sur la programmation en VBA pour EXCEL - Tome 3
- Mémento sur la programmation en VBA pour EXCEL - Tome 4
- Mémento sur la programmation en VBA pour EXCEL - Tome 5
- Mémento sur la programmation en VBA pour EXCEL - Tome 6

I - Introduction

EXCEL possède de multiples fonctions qui permettent de répondre à de nombreux besoins, certes, mais dans certains cas, il n'existe aucune fonction intégrée pour répondre aux besoins particuliers de l'utilisateur. Il faut alors programmer ses propres fonctions…

« Un programme informatique est une séquence d'instructions qui spécifie étape par étape les opérations à effectuer pour obtenir un résultat. Pour écrire un programme informatique, on utilise un langage de programmation ».

Pour programmer EXCEL nous allons donc utiliser un langage de programmation : le VBA, acronyme anglais de « Visual Basic for Applications ».

- Visual : car c'est une « programmation visuelle », grâce à l'éditeur intelligent qui reconnaît les mots clés du langage et permet le débogage.
- BASIC : « Beginner's All Purpose Symbolic Instructions Code » que l'on traduit par « code d'instructions symboliques à tout faire pour débutant ». VBA est un langage de programmation qui se veut simple à apprendre et à utiliser, à la portée de tous.
- Applications : car ce langage de programmation sera commun à la suite bureautique de Microsoft : EXCEL, ACCESS, WORD, POWERPOINT, OUTLOOK, même si chacune de ces applications a ses particularités. Nous allons étudier le VBA utilisé pour programmer EXCEL, mais nous verrons aussi comment, depuis EXCEL, nous pouvons intervenir dans OUTLOOK ou ACCESS, grâce au VBA.

Il faudrait des centaines de pages de documentation pour explorer le VBA dans son intégralité, ici nous allons juste découvrir les bases de la programmation en VBA : comprendre la logique de construction d'un programme, générer des algorithmes, utiliser les principales instructions du langage.
À l'appui d'exemples, d'abord simples puis qui se complexifient graduellement, nous allons aborder de nombreuses notions de la programmation en VBA.
Jusqu'à devenir de véritables champions, capables de rédiger un algorithme de tri rapide. Exercice plus délicat qu'il n'y paraît, car chaque instruction doit être optimisée pour gagner en rapidité de traitement.

Armé de ce bagage, vous pourrez alors approfondir vos connaissances en lisant des ouvrages plus techniques, en farfouillant sur les sites internet, en échangeant sur des forums de programmation.

N'oubliez pas non plus l'aide disponible dans EXCEL, très fournie, pour laquelle un chapitre est consacré afin de vous apprendre à l'exploiter pleinement.

Nous faisons référence dans ce document au VBA version 7.0 pour EXCEL 2010.
Mais la grande majorité du code est compatible avec la version antérieure.

II - Prérequis

Pour programmer en VBA, il suffit de connaître les notions de base d'EXCEL, savoir ce qu'est un classeur, une feuille, une cellule.

Par exemple, savoir que « A1 » fait référence à la cellule de la première ligne, première colonne, de la feuille active.

Car la programmation en VBA utilise ces mêmes notions, et pour mettre la valeur 15 dans la cellule « A1 » de la feuille active, l'instruction est : `Range("A1").Value = 15`

Dans EXCEL, pour mettre manuellement la valeur 15 dans la cellule « A1 » d'une autre feuille, par exemple « Feuil2 », vous devez d'abord sélectionner cette feuille, puis sélectionner la cellule « A1 » et enfin saisir la valeur 15.
En VBA vous pouvez atteindre une cellule sans l'avoir sélectionnée préalablement, simplement en indiquant son « adresse ». L'instruction est : `Sheets("Feuil2").Range("A1").Value = 15`

La syntaxe respecte la logique suivante : **Workbooks** - **Sheets** - **Range** - **Value = 15**

Si l'objet en amont n'est pas renseigné, **Workbooks** (classeur) ou **Sheets** (feuille), c'est l'objet actif qui est pris par défaut.
C'est pourquoi, pour simplifier le code, les objets en amont ne sont indiqués qu'en cas de besoin.

En VBA, on dit que **Range** est un objet, qui possède des propriétés, dont la propriété **Value** (valeur).

Un objet ne peut pas être modifié. Seules ses propriétés peuvent l'être, et encore, pas toutes, car certaines sont en lecture seule.

Un objet peut avoir une propriété par défaut. Par exemple la propriété **Value** pour l'objet **Range**. Dans ce cas il n'est pas nécessaire de saisir cette propriété. Et `Range("A1") = 15` équivaut à `Range("A1").Value = 15`

À la place de **Range** vous pouvez utiliser **Cells(Ligne, Colonne)**.
Cells désigne toujours une seule cellule, alors que **Range** représente une plage pouvant contenir une ou plusieurs cellules. Nous étudierons plus loin comment utiliser **Range** pour une sélection multicellulaire.

Attention, les coordonnées d'une cellule sont exprimées au format (y, x). La première cellule d'une feuille est la cellule située en ligne 1, colonne 1.

Comme pour l'objet **Range**, la propriété **Value** est la propriété par défaut de **Cells**, nous écrirons donc indifféremment `Cells(1,1) = 15` ou `Cells(1,1).Value = 15`

Vous en savez désormais assez pour programmer.

III - Mes premiers pas en programmation VBA pour EXCEL 2010

Pour notre premier programme, imaginons une cour de récréation où deux enfants discutent bonbons. Le premier demande au second : « Combien as-tu de bonbons dans ta poche ? » Pour le savoir, l'enfant plonge sa main dans sa poche, et compte les bonbons : 1, 2, 3… jusqu'à ce qu'il n'en trouve plus. Et donne la solution : « J'en ai 9 ».

Si l'enfant peut réaliser une telle prouesse, c'est parce qu'il suit un processus logique.

Pour commencer, l'enfant réserve dans l'espace mémoire de son cerveau une variable bonbon, elle sera numérique, et non pas alphabétique, car elle sert à compter des nombres. Pour le moment, cette variable vaut 0.
Vient ensuite un traitement en boucle : tant que la poche n'est pas vide, la variable numérique bonbon est incrémentée de une unité.
Quand la condition pour sortir de la boucle est atteinte, ici c'est quand il n'y a plus de bonbon trouvé, alors le traitement se termine et la réponse peut être donnée.

Transposons cet exemple à EXCEL : ouvrez un nouveau classeur.

Dans les cellules « A1 » à « A9 », mettre un « x ».
La feuille EXCEL représente la poche de l'enfant, et les « x » représentent ses bonbons.
Cliquez sur le menu « Développeur » puis « Visual Basic », ou utilisez le raccourci clavier [Alt] [F11].
Si le menu « Développeur » n'est pas affiché dans votre ruban, il faut l'installer via le menu « Fichier », « Options », « Personnaliser le Ruban », cochez dans la partie de droite : « Développeur ».

L'éditeur qui s'affiche est composé de deux parties. Si ce n'est pas le cas, faire « Affichage », « Explorateur de projets » ou [Ctrl][R] . Vous trouverez dans la partie gauche le détail du « Projet », et dans la partie droite, l'éditeur, pour l'instant vierge, où sera saisi notre code de programmation.

Dans la partie gauche, faire un clic droit pour ouvrir le menu contextuel et choisir « Insertion », « Module ».

Le module créé peut être renommé en appelant la fenêtre des propriétés [F4] après avoir été sélectionné.

Notre code sera donc contenu dans un « module » du « projet » du classeur actif. Nous verrons plus tard l'utilité de programmer dans un formulaire, un module, une feuille, ou un classeur.

Faire un clic gauche dans la partie droite, vierge pour l'instant, pour l'activer. Puis choisir dans le menu « Insertion », « Procédure ».
Un formulaire s'ouvre et nous demande le nom de notre procédure, c'est-à-dire le nom de notre programme.
Notre programme va s'appeler : « CompterMesBonbons ». Donc saisissez « CompterMesBonbons » et validez sur « OK » sans changer les options proposées, type « Sub » et portée « Public ». Nous étudierons plus tard ces options.

L'éditeur a généré automatiquement le texte suivant :

```
Public Sub CompterMesBonbons()

End Sub
```

Le texte en bleu signale que le mot est réservé au langage de programmation. Ainsi, nous pourrons avoir une variable nommée Bonbon, mais impossible d'avoir une variable nommée **Public** ou **End**, car ces mots sont réservés à la programmation.
End Sub indique la fin de notre programme.

Justement, programmons :

Il nous faut en premier lieu déclarer les variables que nous allons utiliser, tout comme l'enfant dans notre exemple s'est réservé dans son espace mémoire une variable numérique Bonbon.

C'est le mot **Dim** qui permet cela. Le mot **As** indique le type de la variable. Il existe plusieurs types de variables. Ici nous choisirons un type **Long**. Le choix d'un type de variable est facultatif à notre niveau, mais prenons la bonne habitude de le faire. Nous étudierons ultérieurement les différents types de variables existants.

Il nous faut une deuxième variable Ligne pour passer d'une ligne à l'autre dans la feuille du classeur, pour savoir s'il reste des « x » qui représentent les bonbons. Ainsi nous allons lire la ligne 1, puis la ligne 2, puis la ligne 3 et ainsi de suite, toujours bien sûr dans la colonne 1.

Ce qui donne :

```
Public Sub CompterMesBonbons()

Dim Bonbon As Long
Dim Ligne As Long

End Sub
```

Vous avez remarqué qu'en saisissant **As** une liste de choix s'est ouverte vous permettant de piocher **Long**. Les mots en bleu réservés au langage de programmation peuvent être saisis indifféremment en minuscules ou en majuscules car après validation de la ligne de code, la syntaxe correcte est automatiquement appliquée par l'éditeur.

Quand une variable est déclarée par **Dim** elle est initialisée à zéro.

Parfait pour notre compteur Bonbon, mais nous avons besoin que Ligne soit à 1, car la ligne 0 n'existe pas et essayer de la lire engendrerait une erreur. Cela est fait par le code : Ligne = 1.

Un commentaire peut être inséré avec le guillemet simple « ' ». Le texte s'affiche alors en vert.

```
Public Sub CompterMesBonbons()
```

```
Dim Bonbon As Long
Dim Ligne As Long

Ligne = 1 ' Initialise Ligne à 1

End Sub
```

Maintenant générons notre boucle « Tant que la condition est remplie… Faire ».

VBA étant en anglais, il faudra écrire **While** pour « Tant que ». N'oubliez pas que les coordonnées d'une cellule sont au format (y,x).

Notre boucle débute par **While** suivi de sa condition, et pour indiquer la fin de la boucle **Wend** est utilisé.

La syntaxe officielle est écrite ainsi :

```
While Condition
    [Traitement]
Wend
```

Dans cette syntaxe, [Traitement] représente une ou plusieurs lignes de code.

Pour notre exemple nous allons incrémenter les variables Bonbon et Ligne, avec les instructions « La variable Bonbon égale Bonbon plus un » et « La variable Ligne égale Ligne plus un ».

Il existe d'autres manières pour faire un traitement en boucle, nous les étudierons plus tard.

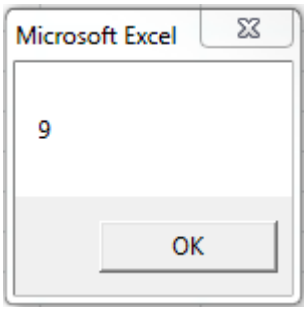
Revenons à notre programme : il nous reste encore à afficher, dans une fenêtre d'information, le contenu de la variable Bonbon à la fin du traitement en boucle, avec l'instruction **MsgBox**. Ne tenez pas compte pour le moment de l'info-bulle qui s'affiche quand vous saisissez cette instruction.

Le code de programmation est :

```
Public Sub CompterMesBonbons()

Dim Bonbon As Long
Dim Ligne As Long

Ligne = 1 ' Initialise Ligne à 1

While Cells(Ligne, 1) <> "" ' Boucle Tant que cellule(y,x) n'est pas vide.
    Bonbon = Bonbon + 1     ' Incrémente le nombre de bonbons.
    Ligne = Ligne + 1       ' Passe à la ligne suivante.
Wend

MsgBox Bonbon ' Affiche le contenu de la variable Bonbon

End Sub
```

Notre programme prend forme. Nous pouvons le tester.

Depuis le classeur, cliquez sur « Macro » dans le menu « Développeur », sélectionnez votre programme puis cliquez sur Exécuter.



Cette fenêtre s'affiche. Vous venez de réaliser votre premier programme en VBA…

Félicitations.

Cliquez sur le bouton « OK ».

Pour suivre l'exécution du programme pas à pas, revenez dans l'éditeur. Placez-vous n'importe où dans le code et faites [F8] ou utilisez le menu « Débogage », « Pas à pas détaillé ».

La ligne en cours d'exécution est mise en surbrillance jaune. Faites [F8] pour passer à la suivante.

Vous suivez en direct le traitement de la boucle « While… Wend ».

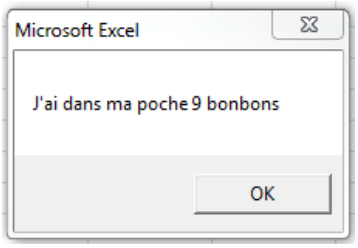
En plaçant la souris sur une variable, sa valeur s'affiche en info-bulle.

Remplacez maintenant l'instruction **MsgBox** Bonbon par :

```
MsgBox "J'ai dans ma poche " & Bonbon & " bonbons"
```

Le signe « & » permet de concaténer le texte placé entre deux doubles guillemets et une variable, ou du texte.

Relancez le programme et vous obtenez maintenant :



EXCEL dimensionne la fenêtre automatiquement.

Maintenant, nous allons personnaliser cette fenêtre, car cet exercice permet d'aborder des notions importantes de VBA.

Lorsque vous saisissez **MsgBox** une info-bulle s'affiche.

```
MsgBox
MsgBox(Prompt, [Buttons As VbMsgBoxStyle = vbOKOnly], [Title], [HelpFile], [Context]) As VbMsgBoxResult
```

Les arguments supportés par l'instruction sont séparés par des virgules. Quand ils sont entre crochets, cela signifie qu'ils sont facultatifs. Vous pouvez sauter un argument facultatif et passer au suivant, mais il faut mettre la virgule demandée. Par exemple : **MsgBox** "Mon Texte", , "Mon Titre" pour ne renseigner que les arguments **Prompt** (message) et **Title** (titre) de la fonction.

Si l'ensemble des arguments est entre parenthèses, c'est que la fonction retourne une valeur. C'est le cas de la fonction **MsgBox** qui retourne une constante, qui correspond au numéro du bouton cliqué par l'utilisateur.

Appuyer sur [F1] pour ouvrir l'aide, très complète, sur cette fonction. Un exemple est disponible.

Ce qu'il faut retenir :

- Prompt : est le texte de la fenêtre. Cet argument est obligatoire. Utilisez deux doubles guillemets pour afficher du texte, comme cela : "Mon Texte". Par contre, les variables, numériques ou alphabétiques, ne doivent pas être entre deux doubles guillemets, sinon c'est le nom de la variable qui est affiché, et non pas sa valeur.
- Buttons : est le style de bouton. Par défaut c'est le style **vbOkOnly**, c'est-à-dire celui qui affiche uniquement le bouton « OK ». Les styles de boutons peuvent être classés en trois catégories :
    - la catégorie qui détermine les boutons à afficher :

vbOKOnly	Affiche le bouton OK uniquement.
vbOKCancel	Affiche les boutons OK et Annuler.
vbAbortRetryIgnore	Affiche les boutons Abandonner, Réessayer et Ignorer.
vbYesNoCancel	Affiche les boutons Oui, Non et Annuler.
vbYesNo	Affiche les boutons Oui et Non.
vbRetryCancel	Affiche les boutons Réessayer et Annuler.

    - La catégorie qui détermine l'icône à afficher :

vbCritical	Affiche l'icône message critique.
vbQuestion	Affiche l'icône question.
vbExclamation	Affiche l'icône message d'avertissement.
vbInformation	Affiche l'icône message d'information.

    - La catégorie qui détermine le bouton sélectionné par défaut :

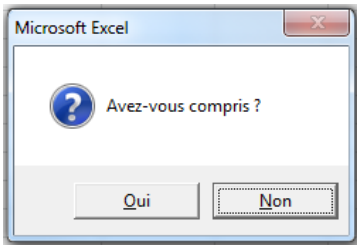
vbDefaultButton1	Le premier bouton est le bouton par défaut.
vbDefaultButton2	Le deuxième bouton est le bouton par défaut.
vbDefaultButton3	Le troisième bouton est le bouton par défaut.
vbDefaultButton4	Le quatrième bouton est le bouton par défaut.

- Title : est le titre de la fenêtre. « Microsoft Excel » s'affichera si cet argument est omis.

Vous pouvez personnaliser une fenêtre en cumulant une des valeurs de chacune de ces catégories de style de bouton.

Par exemple : Buttons = vbYesNo + vbQuestion + vbDefaultButton2 affiche une fenêtre avec les deux boutons « Oui » et « Non », avec une icône symbole d'une question, en sélectionnant le bouton « Non » par défaut.

```
MsgBox "Avez-vous
 compris?", vbYesNo + vbQuestion + v
```

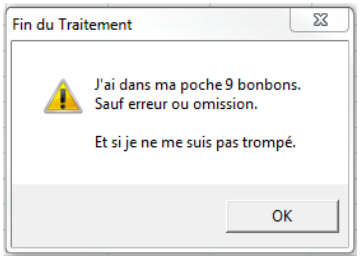


Bon à savoir : le caractère underscore (aussi appelé tiret bas, sous-tiret, blanc souligné, tiret du 8) permet de couper une ligne de code pour plus de lisibilité…

```
MsgBox "Avez-vous compris?", _
        vbYesNo + vbQuestion + vbDefaultButton2
```

Ce qui peut donner aussi :

```
MsgBox "J'ai dans ma
 poche " & Bonbon & "
 bonbons." & Chr(10) & Chr(13) _
        & "Sauf erreur ou
 omission." _
        & Chr(10) & Chr(13) _
        & Chr(10) & Chr(13) _
        & "Et si je ne me suis
 pas trompé.", _

  vbOKOnly + vbExclamation, _
        "Fin du Traitement"
 Affiche le contenu de la
 variable bonbon
```



Note : `Chr(10) & Chr(13)` permet de faire un saut de ligne.

Étudions maintenant comment récupérer le bouton cliqué par l'utilisateur, dans une fenêtre qui permet de faire un choix entre plusieurs boutons.

Nous avons vu que si les arguments sont entre parenthèses la fonction **MsgBox** retourne un chiffre, de type **Long**, qui correspond au numéro du bouton cliqué par l'utilisateur.

Il faut donc prévoir une variable, nommée par exemple Reponse, qui contiendra ce chiffre (Notez que les puristes évitent l'usage des accents dans leurs codes, même si le VBA le permet, pour faciliter la portabilité du code dans d'autres languages de programmation).

Et un traitement conditionnel de la forme : si la condition est remplie alors je fais cela, sinon je fais cela.

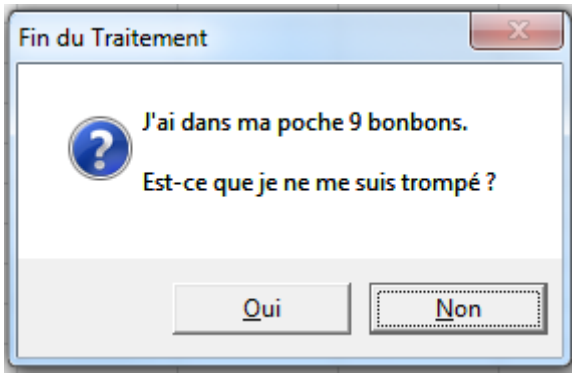
En VBA la syntaxe est la suivante :

```
If Condition Then
    [Traitement]
[Else
    [Traitement]]
End If
```

Modifiez le code de notre programme ainsi et lancez le traitement :

```
Public Sub CompterMesBonbons()
```

```
Dim Bonbon As Long  ' Variable numérique pour compter les bonbons.
Dim Ligne As Long   ' Variable numérique pour balayer les lignes de la feuille.

Ligne = 1 ' Initialise Ligne à 1

While Cells(Ligne, 1) <> "" ' Tant que cellule(y,x) n'est pas vide...
    Bonbon = Bonbon + 1     ' Incrémente le nombre de bonbons,
    Ligne = Ligne + 1       ' Passe à la ligne suivante,
Wend                        ' Boucle.

Dim Reponse As Long  ' Variable numérique pour le choix de l'utilisateur.

Reponse = MsgBox("J'ai dans ma poche " & Bonbon & " bonbons." _
            & Chr(10) & Chr(13) & Chr(10) & Chr(13) _
            & "Est-ce que je ne me suis trompé ?", _
            vbYesNo + vbQuestion + vbDefaultButton2, _
            "Fin du Traitement")

If Reponse = vbYes Then ' Teste si l'utilisateur a choisi le bouton Yes ?
    MsgBox "Mince."     ' Si oui, alors affiche : Mince.
Else                    ' Sinon
    MsgBox "Super."     ' Affiche : Super.
End If                  ' Fin du test conditionnel.

End Sub
```



Deux remarques sur la syntaxe de « If… Then » :

Else est facultatif (d'où la présence des crochets ouverts « [» et fermés «] » dans la syntaxe) vous pouvez faire un test conditionnel de la forme simplifiée :

```
If Condition Then
    [Traitement]
End If
```

Et si le traitement est sur la même ligne que la condition, **End If** n'est plus utile.

```
If Condition Then [Traitement]
```

Nous aurions pu écrire le code suivant, qui donne le même résultat :

```
If Reponse = vbYes Then MsgBox "Mince."     ' Si oui, alors affiche : Mince.
If Reponse <> vbYes Then MsgBox "Super."    ' Si différent de oui, alors affiche : Super.
```

De nos jours, où les ordinateurs sont ultra rapides, avec une mémoire énorme, le choix entre une syntaxe ou une autre relève plus de l'habitude de travailler du programmeur, que d'une recherche de gains en temps de traitement par le microprocesseur.

Même remarque concernant les traitements en boucle.

Nous avons utilisé la syntaxe « Tant que la condition est remplie… Faire » :

```
While Condition
    [Traitement]
Wend
```

Mais nous aurions pu utiliser la syntaxe « Faire… Tant que la condition est remplie » :

```
Do
    [Traitement]
    [Exit Do]
    [Traitement]
Loop [{While | Until} Condition]
```

Ce qui aurait donné le code suivant :

```
Do                                                  ' Faire...
    If Cells(Ligne, 1) = "x" Then Bonbon = Bonbon + 1 ' Si x alors incrémente nombre bonbons,
    Ligne = Ligne + 1                               ' Passe à la ligne suivante,
Loop While Cells(Ligne, 1) <> ""                    ' Boucle Tant Que ligne n'est pas vide.
```

Contrairement à notre première syntaxe avec **While**… **Wend**, où la condition est en amont, et donc s'il n'y a aucun bonbon la boucle n'est pas traitée, **Do**… **Loop While** (ou **Do**… **Loop Until**, **Until** étant l'inverse de **While**, peut être traduit par « Jusqu'à ce que ») exécute le traitement et teste ensuite la condition en fin de boucle. Donc dans le cas où il n'y a aucun bonbon dans la poche, le traitement se déroule quand même. Il faut donc tester si la cellule analysée vaut bien « x » pour incrémenter le nombre de bonbons.

Ou alors faire une boucle avec la syntaxe « Faire tant que la condition est remplie… » :

```
Do [{While | Until} Condition]
    [Traitement]
    [Exit Do]
    [Traitement]
Loop
```

```
Do While Cells(Ligne, 1) = "x"  ' Faire Tant Que cellule vaut x...
    Bonbon = Bonbon + 1 '       ' Incrémente nombre bonbons,
    Ligne = Ligne + 1           ' Passe à la ligne suivante,
Loop                            ' Boucle.
```

Vous avez remarqué que dans certaines syntaxes, vous pouvez sortir de la boucle prématurément en utilisant l'instruction **Exit Do**, généralement déclenchée par un traitement conditionnel de la forme **If… Then**.

L'instruction **Exit Do** peut être utilisée même si une condition de sortie est renseignée.

Enfin, certains préfèrent utiliser la syntaxe « Pour Variable = valeur de début… incrémenter la variable jusqu'à ce qu'elle vaille la valeur de fin » :

```
For VariableCompteur = ValeurDeDébut To ValeurDeFin [Step ValeurDuPas]
    [Traitement]
    [Exit For]
    [Traitement]
Next [VariableCompteur]
```

```
For Ligne = 1 To 9999                       ' Boucle sur les lignes.
    If Cells(Ligne, 1) <> "x" Then Exit For ' Sort de la boucle si
                                            ' la cellule ne vaut pas x.
    Bonbon = Bonbon + 1                     ' incrémente nombre bonbons
Next Ligne                                  ' Passe à la ligne suivante.
```

Si elle n'est pas renseignée, comme ici, ValeurDuPas vaut 1. Mais cette valeur peut être différente, voire négative. Nous utiliserons cette caractéristique bien pratique dans d'autres exemples.

En résumé, il existe plusieurs façons de faire une boucle, libre à vous d'adopter la syntaxe qui vous convient le mieux. Mais dans une boucle, il faut toujours prévoir la sortie, sans quoi vous risquez, soit de tourner en rond sans fin, soit de sortir au mauvais moment (pensez à tester vos boucles avec [F8]). Cependant, pour une meilleur lisibilité du code, évitez si possible l'usage de **Exit Do** et **Exit For**, car cela génère plusieurs sources de sorties et complique le débogage.

Revenons à notre programme, et inspirons-nous de la fonction **MsgBox** qui accepte des arguments, certains obligatoires, d'autres facultatifs, et retourne une valeur. Imaginons que notre procédure puisse en faire autant. Cela lui ferait gagner en souplesse et en puissance… Car notre code pourrait alors être utilisé aussi bien quand la colonne utilisée pour symboliser les bonbons est la colonne 1 que lorsque c'est la colonne 2. Ce qui correspondrait à pouvoir compter les bonbons de la poche droite et ceux de la poche gauche avec le même code. Il suffirait de passer en argument le numéro de la colonne et d'utiliser cet argument. Et la procédure retournerait le nombre de bonbons comptés.

Au travail…

Insérez un nouveau module.

Une procédure qui retourne une valeur n'est pas du type **Sub** (sous-programme en français) mais du type **Function** (fonction en français).

Notre fonction, que l'on va appeler « CompterPoche » aura un argument, MaColonne, de format numérique, qui représentera la colonne de référence, et retournera une valeur numérique qui représentera le nombre de bonbons comptés.

Le code s'inspire de celui déjà étudié, avec en différence, la variable MaColonne qui est utilisée comme coordonnée « x » de la cellule(y,x) à analyser à la place d'une valeur figée à 1 :

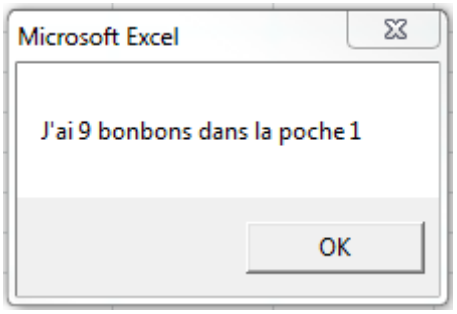
```
Function CompterPoche(MaColonne As Long) As Long

Dim Bonbon As Long  ' Variable numérique pour compter les bonbons.
Dim Ligne As Long   ' Variable numérique pour balayer les lignes de la feuille.

Ligne = 1 ' Initialise Ligne à 1

While Cells(Ligne, MaColonne) <> ""  ' Boucle Tant que cellule(y,x) n'est pas vide.
    Bonbon = Bonbon + 1              ' Incrémente le nombre de bonbons.
    Ligne = Ligne + 1                ' Passe à la ligne suivante.
Wend

CompterPoche = Bonbon ' La fonction retourne le nombre de bonbons comptés

End Function
```

Cette fonction ne peut pas être testée en l'état, avec la touche [F8] de l'éditeur, car elle a besoin que la valeur de son argument MaColonne soit renseignée. Nous ajoutons donc une procédure, dans le même module, de type « sous-programme », nommée « CompterLesBonbons » qui appelle notre fonction avec en argument la colonne désirée. La valeur retournée par la fonction est stockée dans la variable numérique Bonbon.

```
Sub CompterLesBonbons()

Dim Bonbon As Long ' Variable numérique pour les retours.

Bonbon = CompterPoche(1) ' Traitement sur la colonne 1.
MsgBox "J'ai " & Bonbon & " bonbons dans la poche 1"

Bonbon = CompterPoche(2) ' Traitement sur la colonne 2.
MsgBox "J'ai " & Bonbon & " bonbons dans la poche 2"
```

```
End Sub
```

Cette fois vous pouvez tester le code, avec la touche [F8] de l'éditeur, pour obtenir ceci :



Vous avez remarqué que la variable Bonbon est déclarée dans la fonction « CompterPoche » et aussi dans le sous-programme « CompterLesBonbons » : il devrait donc se produire une erreur, car la variable est déclarée deux fois. Or, ce n'est pas le cas.

En effet, une variable déclarée dans une procédure a une utilisation, on dit une portée, limitée à la procédure, ce qui a deux conséquences :

- la variable ne peut être utilisée que dans la procédure où elle est déclarée. Elle est donc unique pour la procédure et même si une variable de même nom est déclarée dans une autre procédure, cette autre variable est physiquement différente. Ça tombe plutôt bien, car ça évite de se casser la tête pour trouver des noms différents à toutes les variables que nous allons utiliser dans un programme ;
- la variable est effacée de la mémoire à la fin de la procédure, son contenu est donc perdu… sauf si elle est déclarée avec **Static** au lieu de **Dim**. Dans ce cas la valeur de la variable est conservée, mais la variable ne peut pas être utilisée dans une autre procédure.

Pour qu'une variable soit commune à toutes les procédures d'un module, elle doit être déclarée en en-tête du module, avant les procédures, avec **Dim**, ou **Private**.

Pour qu'une variable soit commune à toutes les procédures de tous les modules, elle doit être déclarée en en-tête d'un des modules, avant les procédures, avec **Public**.

Les procédures aussi ont une portée :

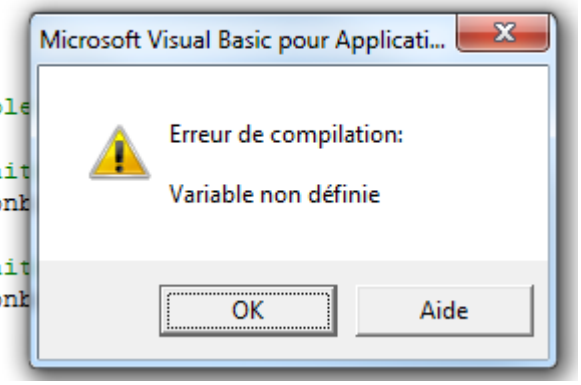
- une procédure déclarée avec le mot clé **Private** ne peut être appelée que depuis une procédure au sein du même module ;
- une procédure déclarée avec le mot clé **Public** peut être appelée depuis toutes les procédures de tous les modules. Les procédures sont **Public** par défaut.

Dernière remarque au sujet de nos variables : elles ne sont pas identifiées par un code de couleur par l'éditeur, au contraire des mots clés du langage qui sont affichés en bleu. Donc il suffit d'une faute de frappe pour que la variable Bombon soit utilisée à la place de la variable Bonbon dans une partie du traitement. Car bien que Bombon ne soit pas déclarée, l'éditeur accepte cette variable, en lui attribuant par défaut le type **Variant** et en l'initialisant à la valeur nulle. De quoi mettre un beau bazar dans un programme et des migraines en perspective pour trouver l'origine du bogue.

Pour se prémunir de ce danger, nous allons prendre l'habitude d'obliger la déclaration des variables, en écrivant **Option Explicit** en en-tête du module, avant les autres déclarations de variables et de procédures.

Ainsi, en cas d'erreur de frappe, l'éditeur bloque le traitement quand l'on veut exécuter la procédure :

```
Option Explicit

Sub CompterLesBonbons()

Dim Bonbon As Integer ' Variable

Bonbon = CompterPoche(1) ' Trait
MsgBox "J'ai " & Bombon & " bonb

Bonbon = CompterPoche(2) ' Trait
MsgBox "J'ai " & Bonbon & " bonb

End Sub
```



Vous pouvez forcer cette déclaration depuis l'éditeur : menu Outils, puis Options, cochez « Déclaration des variables obligatoire ».

De toute façon, en y regardant de plus près, nous n'avons pas besoin de la variable Bonbon dans notre programme !

En effet, dans le sous-programme « CompterLesBonbons » cette variable est utilisée pour stoker le retour de l'appel à la fonction « CompterPoche » et sert ensuite à former le message à afficher avec **MsgBox**.

Mais le message peut être construit directement avec l'appel à la fonction « CompterPoche » :

```
Sub CompterLesBonbons()

MsgBox "J'ai " & CompterPoche(1) & " bonbons dans la poche 1"
MsgBox "J'ai " & CompterPoche(2) & " bonbons dans la poche 2"

End Sub
```

De même dans la fonction « CompterPoche », la variable Bonbon sert de compteur provisoire dont on peut se passer en incrémentant directement « CompterPoche », sachant que sa valeur de retour est initialisée à zéro à chaque appel. La fonction se transforme alors en une variable :

```
Function CompterPoche(MaColonne As Long) As Long

Dim Ligne As Long ' Variable numérique pour balayer les lignes de la feuille.
Ligne = 1         ' Initialise Ligne à 1.

While Cells(Ligne, MaColonne) <> "" ' Tant que cellule(y,x) n'est pas vide.
    CompterPoche = CompterPoche + 1 ' Incrémente le nombre de bonbons.
    Ligne = Ligne + 1               ' Passe à la ligne suivante.
Wend                                ' Boucle

End Function
```

Le code de notre programme est maintenant plus compact. Mais est-il plus lisible pour autant ? Le programmeur ne doit pas oublier que son programme aura peut-être besoin d'être relu et compris par une autre personne.

Nous sommes fiers de notre programme, mais imaginons que maintenant, nous devions aussi compter le nombre de scoubidous et de bonbons cachés dans le cartable de notre enfant.

Sur la feuille EXCEL, dans les cellules « C1 » à « C9 », mettez aléatoirement un « x » pour symboliser les bonbons et un « s » pour symboliser les scoubidous. Évidemment, la colonne 3 symbolise le cartable.

La fonction « CompterPoche » peut servir de base : car si actuellement, elle considère qu'une cellule non vide représente un bonbon et qu'elle ne permet donc pas de distinguer les bonbons des scoubidous, il suffit d'ajouter un

argument alphabétique, nommé MonChoix, pour savoir si l'on doit compter des bonbons « x » ou des scoubidous « s » ainsi qu'une condition qui compare la valeur de la cellule avec cet argument.

Enfin, je veux une nouvelle procédure, indépendante de la procédure « CompterLesBonbons », pour compter les éléments du cartable.

Le code de la fonction devient :

```
Function CompterPoche(MaColonne As Long, MonChoix As String) As Long

Dim Ligne As Long    ' Variable numérique pour balayer les lignes de la feuille.

Ligne = 1 ' Initialise Ligne à 1

While Cells(Ligne, MaColonne) <> ""             ' Tant Que cellule(y,x) n'est pas vide...
    If Cells(Ligne, MaColonne) = MonChoix Then  ' Si la cellule est ce qu'il faut compter,
        CompterPoche = CompterPoche + 1         ' Incrémente le nombre d'items,
    End If                                      ' Fin de la condition.
    Ligne = Ligne + 1                           ' Passe à la ligne suivante.
Wend                                            ' Boucle.

End Function
```

Et je peux écrire ma nouvelle procédure « CompterLeCartable » :

```
Sub CompterLeCartable()

MsgBox "J'ai " & CompterPoche(3, "x") & " bonbons et " _
       & CompterPoche(3, "s") & " scoubidous dans mon cartable."

End Sub
```

Mais il faut aussi modifier le sous-programme « CompterLesBonbons » que nous avons écrit, en ajoutant le nouvel argument requis pour l'appel de la fonction « CompterPoche » :

```
Sub CompterLesBonbons()

MsgBox "J'ai " & CompterPoche(1, "x") & " bonbons dans la poche 1"
MsgBox "J'ai " & CompterPoche(2, "x") & " bonbons dans la poche 2"

End Sub
```

Et c'est cela le problème. Je dois modifier une procédure qui marche bien, pour ajouter un argument qui à première vue, ne sert pas à grand-chose. Il faut donc trouver une autre solution pour que l'appel à « CompterPoche » reste compatible…

C'est possible, en déclarant facultatif le nouvel argument MonChoix, comme le fait l'instruction **MsgBox** que nous avons étudiée, avec ses arguments Buttons, Title…

La procédure gère alors deux situations :

- soit MonChoix n'est pas renseigné et est une chaîne vide, dans ce cas « CompterPoche » est incrémenté ;
- soit MonChoix est renseigné par « x » ou « s » et dans ce cas « CompterPoche » est incrémenté uniquement si la cellule contient la valeur de l'argument MonChoix.

Dit autrement, si MonChoix est vide ou si la cellule analysée vaut MonChoix, alors « CompterPoche » est incrémentée.

Nous ne l'avions pas encore vu, mais une « *Condition* » peut être constituée de plusieurs expressions. Généralement on combine les opérateurs logiques OR (ou bien) et AND (et aussi) pour générer une condition multicritères.

Le code modifié permet à la procédure « CompterLesBonbons » de rester compatible avec l'ancien appel de « CompterPoche », car le nouvel argument est facultatif :

```
Function CompterPoche(MaColonne As Long, Optional MonChoix As String) As Long

Dim Ligne As Long
Ligne = 1

While Cells(Ligne, MaColonne) <> ""

    If MonChoix = "" Or Cells(Ligne, MaColonne) = MonChoix Then
        CompterPoche = CompterPoche + 1
    End If

    Ligne = Ligne + 1
Wend

End Function
```

Super, notre nouvelle procédure marche à merveille. Mais elle a un inconvénient. Pour compter les éléments du cartable, il faut appeler deux fois la fonction. Une fois pour compter les bonbons, une autre fois pour compter les scoubidous. Si nous pouvions ne l'appeler qu'une seule fois… le temps de traitement serait divisé par deux. Ça peut valoir la peine d'étudier une autre solution…

Souvenez-vous de notre fonction d'origine :

```
Function CompterPoche(MaColonne As Long) As Long

Dim Ligne As Long ' Variable numérique pour balayer les lignes de la feuille.
Ligne = 1         ' Initialise Ligne à 1.

While Cells(Ligne, MaColonne) <> "" ' Tant que cellule(y,x) n'est pas vide.
    CompterPoche = CompterPoche + 1 ' Incrémente le nombre de bonbons.
    Ligne = Ligne + 1               ' Passe à la ligne suivante.
Wend                                ' Boucle

End Function
```

Nous allons ajouter à cette fonction deux arguments facultatifs de type numérique : NbBonbon qui sera incrémenté quand la cellule analysée vaut « x », et NbScoubidou qui sera incrémenté quand la cellule analysée vaut « s ».

Nous avions déjà utilisé plusieurs fois la notion d'argument, facultatif ou non, mais jusqu'à présent l'argument était passé en dur : il valait 1, 2, 3, « x » ou « s ». Ici nous allons passer en argument une variable, préalablement déclarée dans la procédure « CompterLeCartable ». En VBA, cette variable est passée par référence (sauf si le mot clé **ByVal** est explicitement déclaré) et peut donc être modifiée.

Concrètement, cela signifie que la procédure appelée peut modifier la valeur de la variable passée en argument. Et la procédure appelante peut utiliser la nouvelle valeur de la variable ainsi modifiée.

Vous allez mieux comprendre en exécutant la procédure « CompterLeCartable » :

```
Sub CompterLeCartable()

Dim MesBonbon As Long    ' Variable numérique qui contiendra le nombre de bonbons
Dim MesScoubidou As Long ' Variable numérique qui contiendra le nombre de scoubidous

Call CompterPoche(3, MesBonbon, MesScoubidou)  ' Alimente MesBonbon et MesScoubidou

MsgBox "J'ai " & MesBonbon & " bonbons et " _
       & MesScoubidou & " scoubidous dans mon cartable."

End Sub
```

Notez que la fonction « CompterPoche » est conçue pour retourner une variable, qui n'est pas utilisée ici. C'est pourquoi il faut faire précéder l'appel à la fonction du mot clé **Call** pour prévenir VBA que nous faisons délibérément un appel simple, sans stocker la valeur retournée par la fonction.

La fonction « CompterPoche » a été modifiée ainsi :

```
Function CompterPoche(MaColonne As Long, _
                      Optional NbBonbon As Long, _
                      Optional NbScoubidou As Long) As Long

Dim Ligne As Long     ' Variable numérique pour balayer les lignes de la feuille.
Ligne = 1             ' Initialise Ligne à 1

While Cells(Ligne, MaColonne) <> ""  ' Tant que cellule(y,x) n'est pas vide.

    CompterPoche = CompterPoche + 1 ' Incrémente le nombre d'éléments trouvés.
    If Cells(Ligne, MaColonne) = "x" Then NbBonbon = NbBonbon + 1       ' Les bonbons.
    If Cells(Ligne, MaColonne) = "s" Then NbScoubidou = NbScoubidou + 1 ' Les scoubidous.
    Ligne = Ligne + 1 ' Passe à la ligne suivante.

Wend

End Function
```

Le plus déroutant pour un débutant est que les noms des variables sont différents, alors qu'ils représentent les mêmes portions de mémoire pour l'ordinateur. Par exemple, MesBonbon est la variable utilisée comme premier argument dans l'appel de la fonction « CompterPoche », alors que cette dernière va utiliser NbBonbon dans ses traitements, et donc finalement, modifier la valeur de MesBonbon !

Rassurez-vous, vous parviendrez rapidement à jongler avec ces subtilités.

Les arguments d'une fonction peuvent aussi être nommés. La syntaxe des arguments nommés est :

```
NomArgument  := Valeur
```

Dans ce cas, l'ordre de passage des arguments est libre.

L'appel à la fonction « CompterPoche » peut alors prendre cette forme :

```
Call CompterPoche(3, NbScoubidou:=MesScoubidou, NbBonbon:=MesBonbon)
```

Cette forme d'appel est cependant rarement utilisée, car elle nécessite plus de temps de rédaction, mais la lecture est beaucoup plus claire. La généralisation de cette pratique simplifierait pourtant la maintenance des programmes.

Une nouvelle façon d'aborder le problème…

Il faut déclarer en en-tête du module les variables MesBonbon et MesScoubidou, pour qu'elles puissent être utilisées par toutes les procédures du module.

Et alimenter ces variables dans la fonction « CompterPoche » qui du fait n'a plus besoin d'argument facultatif.

Il faut cependant prendre soin de mettre les variables à zéro avant d'appeler la fonction, au cas où elles auraient déjà été utilisées (par d'autres fonctions ou par cette fonction), ce qui fausserait les résultats.

```
Option Explicit

Dim MesBonbon As Long    ' Variable numérique qui contiendra le nombre de bonbons
Dim MesScoubidou As Long ' Variable numérique qui contiendra le nombre de scoubidous

Sub CompterLeCartable()
```

```
Call CompterPoche(3)  ' Alimente MesBonbon et MesScoubidou

MsgBox "J'ai " & MesBonbon & " bonbons et " _
        & MesScoubidou & " scoubidous dans mon cartable."

End Sub

Function CompterPoche(MaColonne As Long) As Long

Dim Ligne As Long               ' Variable numérique pour balayer les lignes de la feuille.
Ligne = 1                       ' Initialise Ligne à 1
MesBonbon = 0: MesScoubidou = 0 ' Variables mises à zéro par sécurité.

While Cells(Ligne, MaColonne) <> ""  ' Tant Que cellule(y,x) n'est pas vide.

    CompterPoche = CompterPoche + 1 ' Incrémente le nombre d'éléments trouvés.
    If Cells(Ligne, MaColonne) = "x" Then MesBonbon = MesBonbon + 1          ' Les bonbons.
    If Cells(Ligne, MaColonne) = "s" Then MesScoubidou = MesScoubidou + 1    ' Les scoubidous.
    Ligne = Ligne + 1 ' Passe à la ligne suivante.

Wend

End Function
```

Notez l'usage du symbole « : » (deux points) qui permet de coder plusieurs instructions sur une même ligne.

Cette dernière méthode est peut-être la plus simple. Comme toujours en programmation, il y a plusieurs manières de faire. C'est à chacun de s'approprier la méthode qui lui convient le mieux, tout en restant en adéquation avec les nécessités de l'application à développer.

Avant de continuer je vous propose un peu de théorie sur les variables, car ces notions nous seront nécessaires par la suite.

IV - Les différents types de variables

L'aide d'EXCEL donne un tableau complet, ici je ne reprends que les types couramment utilisés :

Type de données	Taille en mémoire	Quand utiliser ce type de données
Byte	1 octet	Pour un compteur allant de 0 à 255.
Boolean	2 octets	Pour retourner Vrai **True** (-1) ou Faux **False** (0).
Integer	2 octets	Pour un compteur allant -32 768 à 32 767.
Long	4 octets	Pour un compteur allant -2 147 483 648 à 2 147 483 647.
Double	8 octets	Pour gérer des données à virgule.
Currency	8 octets	Pour gérer des devises, avec 4 chiffres après la virgule.
Date	8 octets	Pour les dates, du 1er janvier 100 au 31 décembre 9999.
String	Longueur de la chaîne	Pour les chaînes de caractères. 1 à environ 2 milliards (2^{31}) caractères.
Variant	16 octets ou 24 octets	Quand on ne sait pas quel type sera retourné par la fonction appelée ou que la fonction peut retourner Null, False, True.

Rappel des évidences :

- vous devez toujours adapter le type des variables utilisées à vos besoins ;
- plus vous minimisez les ressources utilisées, plus vos traitements tourneront rapidement ;
- un dépassement de capacité provoque un plantage de l'application.

Cas particulier : une chaîne de caractères peut avoir une longueur fixe, par exemple 20 caractères : `Dim Nom As String * 20`

Type défini par l'utilisateur :

Ce type est utilisé pour la déclaration de variables contenant plusieurs types d'information, qui seront ainsi regroupés. Par exemple une couleur est composée des éléments rouge, vert, bleu.

Nous pouvons, dans un premier temps, définir un type « Couleur » contenant ces trois éléments.

```
Type Couleur          ' Déclaration d'un type de données personnel
    Rouge As Long     ' Définition du 1er élément
    Vert As Long      ' Définition du 2e élément
    Bleu As Long      ' Définition du 3e élément
End Type              ' Fin de la déclaration
```

Puis, dans un second temps, créer une variable « MaCouleur » de type « Couleur ».

```
Dim MaCouleur As Couleur ' Création d'une variable MaCouleur de type "Couleur"
```

L'appel des éléments de la variable « Couleur » se fait en mettant un point après la variable :

```
Sub Test()
Dim MaCouleur As Couleur

MaCouleur.Rouge = 0
MaCouleur.Vert = 255
MaCouleur.Bleu = 0

End Sub
```

V - Les tableaux de variables

Un tableau est une variable comportant plusieurs éléments. On accède à un élément en utilisant un indice.

Le premier élément de la variable est l'indice 0. Dim MonTableau(i) dimensionne un tableau des indices 0 à i soit i +1 éléments. Sauf si l'on utilise l'instruction Option Base 1, dans ce cas le premier élément est l'indice 1. Mais fuyez cette instruction qui déroge aux habitudes.

Pour prévenir toute confusion, l'usage est de spécifier explicitement la plage d'indice pour la variable, en utilisant la syntaxe **(**IndiceMini **To** IndiceMaxi**)**.

Les règles de portée des tableaux sont les mêmes que pour les variables.

Par exemple, ce tableau peut stocker le nom de 20 personnes :

```
Dim MonTableau(0 To 19) As String ' Déclare un tableau de 20 éléments
MonTableau(0) = "Toto" 'Initialise le 1er élément
MonTableau(1) = "Titi" 'Initialise le 2e élément
```

Un tableau peut avoir jusqu'à 60 dimensions. Chaque dimension a une plage d'indices qui lui est propre. L'exemple suivant permet de stocker la valeur des 100 cellules d'un carré de 10 sur 10, dans un tableau à deux dimensions :

```
Sub MémoriseUnCarré()

Dim MaCellule(1 To 10, 1 To 10) As Variant ' Tableau de 10 éléments à deux dimensions.
Dim y As Integer, x As Integer

For y = 1 To 10      ' Boucle sur les 10 premières lignes.
    For x = 1 To 10 ' Boucle sur les 10 premières colonnes.
        MaCellule(y, x) = Cells(y, x) ' Mémorise le contenu de la cellule
    Next x           ' Colonne suivante.
Next y               ' Ligne suivante

End Sub
```

Un tableau de données peut avoir des indices négatifs : Dim MonTableau(-4 To 7) As Integer

Indice :	-4	-3	-2	-1	0	1	2	3	4	5	6	7
Valeur :	**3**	**9**	**5**	**5**	**15**	**7**	**8**	**20**	**17**	**0**	**5**	**90**
Elément :	1	2	3	4	5	6	7	8	9	10	11	12

Le premier élément du tableau vaut 3 et son indice est -4. L'indice 6 représente le 11^{e} élément du tableau, sa valeur est 5.

Le mot clé **Lbound**(NomDuTableau), donne le numéro de l'indice le plus petit du tableau.
Le mot clé **Ubound**(NomDuTableau), donne le numéro de l'indice le plus grand du tableau.

Un tableau peut être dynamique, c'est-à-dire que sa dimension n'est pas définie lors de la déclaration de la variable, mais au cours du traitement, suivant les besoins nécessaires. Cela permet l'optimisation de la mémoire :

- Le mot clé ReDim Preserve NomDuTableau(i To n), indique qu'il faut redimensionner le tableau de i à n, en conservant les données déjà existantes.

> ⚠ Attention : pour les tableaux à plusieurs dimensions, seule la dernière dimension peut être redimensionnée.

- Le mot clé ReDim NomDuTableau(i To n), indique qu'il faut redimensionner le tableau de i à n, sans conserver les données existantes. Les données sont alors toutes mises à zéro.

Un tableau de type variant peut être alimenté rapidement : MonTableau = Array("Toto", "Titi", ...)
Un tableau à plusieurs dimensions peut aussi être dynamique.
Les règles de portée des tableaux dynamiques sont les mêmes que pour les variables.

Par exemple pour mémoriser la valeur des cellules de la colonne A, en sachant qu'une cellule vide indique la fin des cellules à mémoriser, il faut utiliser un tableau dynamique, car nous ne savons pas à l'écriture du code combien il y aura d'éléments à mémoriser.

```
Sub MémoriseColonneA()

Dim MaCellule() As Variant ' Déclare un tableau dynamique à une dimension.
Dim y As Long              ' Variable qui indique la ligne à analyser.
Dim i As Long              ' Variable qui dimensionne le tableau.

y = 1
While Cells(y, 1) <> ""          ' Boucle sur les lignes de la colonne A.
    ReDim Preserve MaCellule(i)  ' Redimensionne le tableau MaCellule sans l'effacer.
    MaCellule(i) = Cells(y, 1)   ' Mémorise la valeur de la cellule.
    i = i + 1                    ' Incrémente l'indice du tableau.
    y = y + 1                    ' Passe à la ligne suivante.
Wend

' Boucle sur les éléments du tableau pour afficher
' l'indice et la valeur de l'élément.
For i = LBound(MaCellule) To UBound(MaCellule)
    MsgBox "Indice : " & i & Chr(10) & Chr(13) & "Valeur : " & MaCellule(i)
Next i

End Sub
```

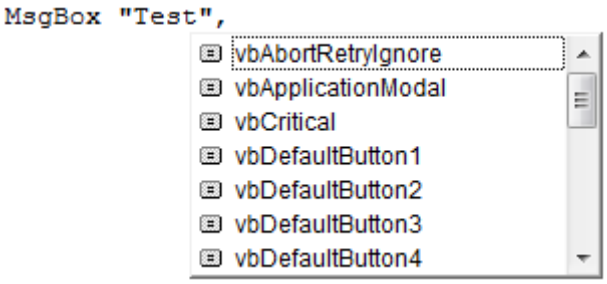
Pour être presque complet sur le sujet des variables, il faut parler des constantes et des énumérations.

Une constante représente une variable, numérique ou de type chaîne, qui ne peut pas être modifiée. Sa valeur reste donc constante.
Le mot clé **Const** permet de déclarer une constante et de définir sa valeur fixe.
Les règles de portée des constantes sont les mêmes que pour les autres variables.
Généralement, une constante est déclarée **Public** dans l'en-tête de module pour pouvoir être utilisée par toutes les procédures de l'application.

Exemple de déclaration :

```
Public Const MaVersion = "Version 1.0" ' Constante de type chaîne
Public Const MaCouleur = 15 ' Constante de type numérique
```

L'énumération permet d'attribuer des noms à des valeurs numériques, de la même manière que les constantes, à la différence que ces noms sont regroupés dans un ensemble. Les énumérations sont peu utilisées, pourtant elles peuvent rendre de précieux services dans l'utilisation de nos procédures, à la manière de l'instruction **MsgBox** qui affiche une liste déroulante des arguments disponibles lorsque l'on commence la saisie :



L'énumération est généralement déclarée **Public** dans l'en-tête de module pour pouvoir être utilisée par toutes les procédures de l'application.
Le mot clé **Enum** permet de déclarer une énumération, et ses éléments.
Le mot clé **End Enum** doit être utilisé pour indiquer la fin de l'énumération.
Les valeurs des constantes sont toujours de type **Long**, donc un entier positif ou négatif. Elles peuvent être omises, mais dans ce cas le premier membre aura pour valeur 0. Les autres membres auront la valeur du membre précédent plus 1.

Dans l'exemple qui suit, nous déclarons de type **Enum** une variable nommée MesCouleurs et affectons une valeur à chacun de ses éléments :

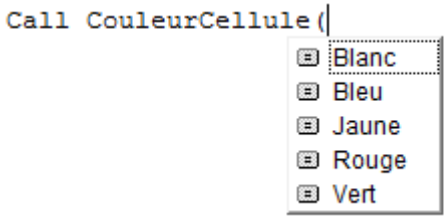
```
Public Enum MesCouleurs
    Bleu = 8210719
    Blanc = 16777215
    Rouge = 255
    Vert = 5287936
    Jaune = 65535
End Enum
```

Puis nous créons une procédure « CouleurCellule » qui change la couleur de fond de la cellule active.

L'argument ChoixCouleur est une variable qui indique la couleur à utiliser. Nous déclarons cette variable de type MesCouleurs :

```
Sub CouleurCellule(ChoixCouleur As MesCouleurs)
    ActiveCell.Interior.Color = ChoixCouleur
End Sub
```

Ainsi, lorsque l'on appelle cette fonction, une liste déroulante des éléments disponibles s'ouvre pour permettre au programmeur de piocher la couleur de son choix :



Vous remarquerez que la liste déroulante classe automatiquement par ordre alphabétique croissant les éléments disponibles.

La technique est très facile à mettre en place, et elle sécurise les développements, tout en restant souple. Car il reste possible de faire un appel à la fonction en passant en argument une valeur différente des éléments proposés, ou de passer en argument une variable.
De plus, il suffit de modifier la valeur de la constante (une seule fois) pour que la modification se répercute à l'ensemble du code.

Exemples d'appel avec une valeur déterminée ou avec une variable :

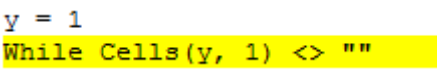
```
Call CouleurCellule(0)
Call CouleurCellule(MaCouleur)
```

VI - L'éditeur VBA

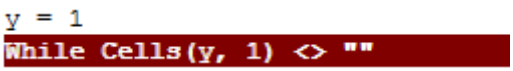
Nous utilisons l'éditeur VBA pour écrire notre code. grâce à son moteur intelligent, il reconnaît les mots clés du langage et les affiche en bleu. Les commentaires sont affichés en vert.
Vous pouvez changer ces paramètres de l'éditeur dans le menu « Outils », puis « Options ».

L'éditeur permet aussi de suivre le déroulement d'un traitement, explorer la valeur des variables. C'est le débogage. Nous avons déjà vu que la touche [F5] permet d'exécuter une procédure, et que la touche [F8] permet de suivre le traitement pas à pas. L'éditeur offre d'autres possibilités, dont voici les principales :

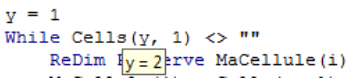
[Ctrl][F8] : exécute jusqu'au curseur. La ligne est en surbrillance jaune.



[Ctrl][F9] : placer un point d'arrêt. La ligne est en surbrillance rouge. Le traitement s'arrêtera à cette ligne.

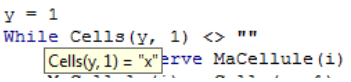


Placer la souris sur une variable : affiche dans une info-bulle la valeur de la variable.



Vous pouvez aussi sélectionner la variable et faire [Majuscule][F9]

Placer la souris sur une instruction : affiche dans une info-bulle la valeur de l'instruction.

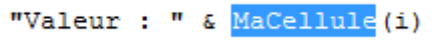


Vous pouvez aussi sélectionner l'instruction et faire [Majuscule][F9]

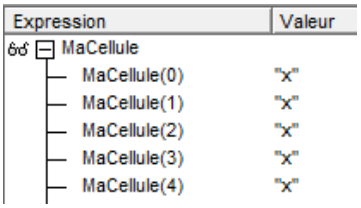
Ajouter un espion sur une valeur sélectionnée : clic droit, puis « ajouter un espion ».

La valeur de l'expression reste affichée en permanence.

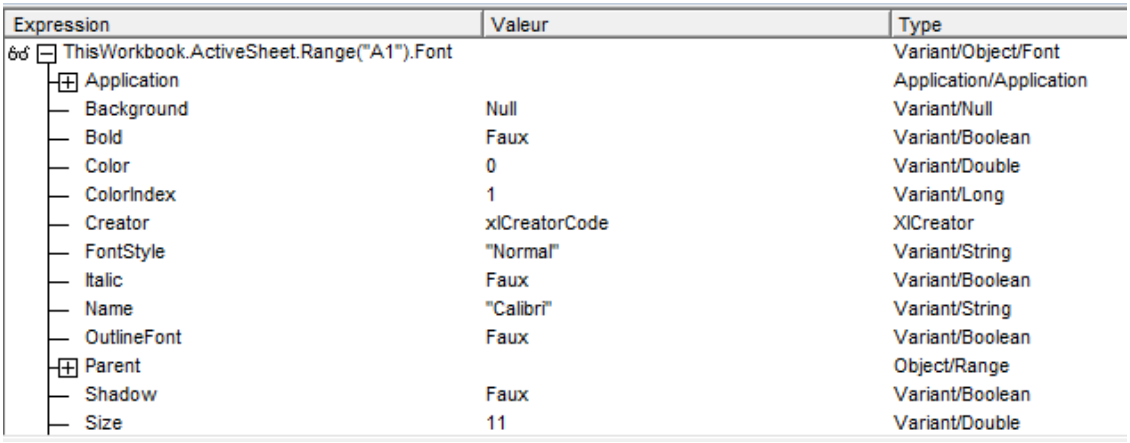
C'est valable aussi pour un tableau, en ne sélectionnant que le tableau et non pas l'ensemble tableau plus indice :



Le tableau n'a pas de valeur, il faut cliquer sur le + pour voir les différents éléments :



C'est valable aussi pour certains objets, ce qui permet de connaître les différentes propriétés de l'objet :



Pensez aussi à la fenêtre d'exécution qui s'ouvre avec [Ctrl][G] :

Elle permet d'interroger la valeur d'une variable, ou d'une expression, précédée d'un point d'interrogation. Par exemple pour connaître la valeur de la variable y ou la couleur de fond de la cellule « A1 » :

```
Exécution
? y
 2
? Range("A1").font.color
 0
```

Ou elle permet d'afficher la valeur d'une variable avec l'instruction Debug.Print Variable placée dans le code. Dans notre procédure « MémoriseColonneA » nous affichons la valeur des cellules dans un message, nous pouvons aussi l'afficher dans la fenêtre d'exécution :

```
For i = LBound(MaCellule) To UBound(MaCellule)
    Debug.Print "Indice : " & i & " Valeur : " & MaCellule(i)
Next i
```

```
Exécution
 Indice : 0  Valeur : x
 Indice : 1  Valeur : x
 Indice : 2  Valeur : x
 Indice : 3  Valeur : x
 Indice : 4  Valeur : x
 Indice : 5  Valeur : x
 Indice : 6  Valeur : x
```

Plus fort encore, la fenêtre d'exécution permet de lancer une procédure, une fonction ou une instruction :

Exécution
```
Call MémoriseColonneA
```

Lorsqu'une partie du code a un intérêt particulier, vous pouvez y placer un signet, et vous déplacer rapidement d'un signet à l'autre :

Icônes pour placer un signet, aller au suivant, retourner au précédent, effacer les signets.

Pour une saisie rapide de votre code, abusez de [Ctrl][J], qui ouvre la liste des mots clés, des variables, des propriétés, des méthodes et procédures, et sélectionne le mot désiré à mesure de votre saisie.

Enfin, avant de tester un traitement, testez déjà sa syntaxe par le menu « Débogage », « Compiler VBAProjet ».

VII - Exercice pratique : Créer une procédure de tri des données

Pour mettre en pratique une grande partie de ce que nous venons d'étudier sur la programmation en VBA, je vous propose un exercice qui va permettre de se faire la main sur les algorithmes.

Sur une feuille EXCEL mettre dans les cellules « A1 » à « A5 » les valeurs suivantes : 5, 6, 2, 1, 3.

Nous allons créer une première procédure pour récupérer ces données, et une seconde pour les trier par ordre croissant ou décroissant. Bien sûr, cette procédure devra marcher quel que soit le nombre de données à trier.

Enfin nous allons restituer ces données triées en « B1 » à « B5 ».

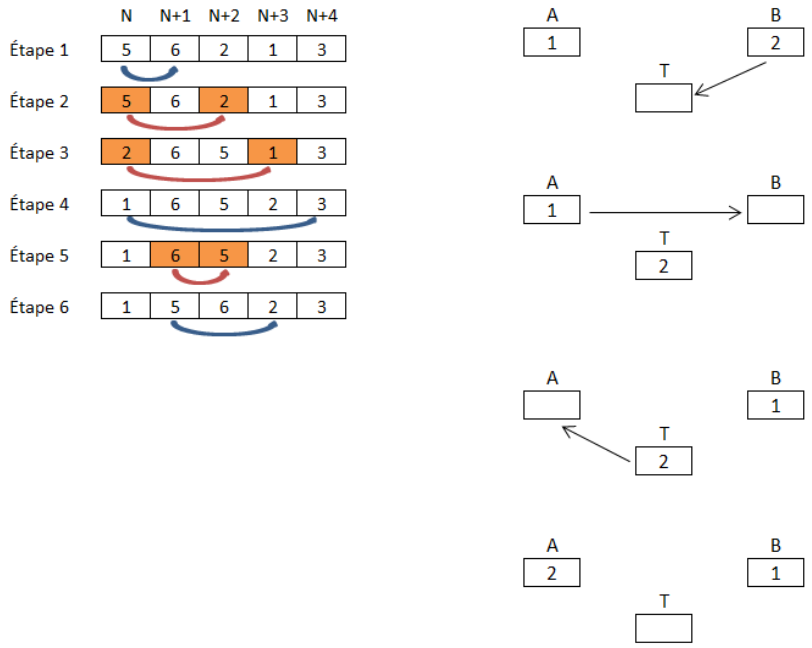
Récupérer les données de la feuille pour les charger dans un tableau, et inversement, afficher en colonne B les données d'un tableau, nous savons faire. Par contre il nous faut un algorithme pour trier les données.

Avant d'aller voler sur internet un algorithme scientifique comme QuickSort, essayons de créer notre propre algorithme, histoire de faire travailler nos méninges…

Le plus simple est de faire une boucle sur les données. Donc ici de N à N+4. Comparer si N+1 est inférieure à N (dans le cas d'un tri par ordre croissant, ou supérieure pour un ordre décroissant). Dans ce cas, il faut intervertir les données N+1 et N. Et continuer avec la comparaison entre N et N+2… Jusqu'à N+4.

En fin de boucle l'on repart de N+1 à N+4.
Et ainsi de suite.

Ce qui donne la représentation suivante :



L'échange entre A et B

Et ainsi de suite…

Nous avons deux boucles dans notre algorithme, l'une imbriquée dans l'autre :

```
For n = 0 To 4
    For i = n + 1 to 4
        ' Comparaison de n et i, avec échange des données si besoin selon l'ordre désiré.
        ' L'échange des données nécessite trois traitements comme démontré ci-joint.
    Next i
Next n
```

La procédure de chargement des données, d'appel de la fonction de tri, et d'affichage du résultat :

```
'-------------------------------------------------------------------------------
Sub ChargeLesDonnées()
'-------------------------------------------------------------------------------
Dim MonTableau() As Variant ' Déclare un tableau Variant dynamique.
Dim y As Long               ' Indique la ligne à analyser.
Dim i As Long               ' Indice du tableau.

y = 1
While Cells(y, 1) <> ""           ' Boucle sur les lignes de la colonne A.
    ReDim Preserve MonTableau(i)  ' Redimensionne le tableau d'après l'indice.
    MonTableau(i) = Cells(y, 1)   ' Mémorise la valeur de la cellule.
    i = i + 1                     ' Incrémente l'indice du tableau.
    y = y + 1                     ' Passe à la ligne suivante.
Wend

Call TriDonnées(MonTableau(), True)    ' Appelle la fonction de tri du tableau.

For y = LBound(MonTableau) To UBound(MonTableau) ' Boucle sur les éléments.
    Cells(y + 1, 2) = MonTableau(y)              ' Affiche l'élément en B.
Next y

Application.ScreenUpdating = True

End Sub
```

La procédure avec notre algorithme de tri :

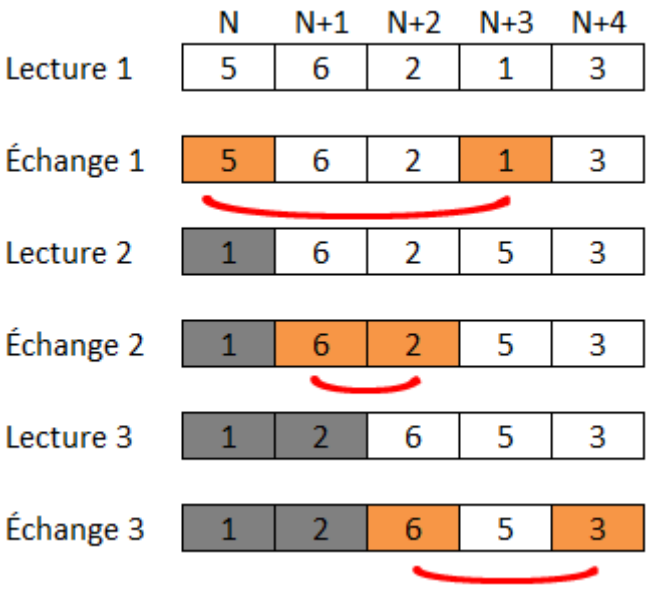
```
'-------------------------------------------------------------------------------
Function TriDonnées(ByRef TabDonnées() As Variant, _
                    Optional OrdreCroissant As Boolean = True)
'-------------------------------------------------------------------------------
' Procédure pour trier par ordre croissant ou décroissant les données passées
' en argument dans un tableau à une dimension. Le tableau est actualisé.
' Le tableau étant de type variant les chaînes et les numériques sont gérés.
'-------------------------------------------------------------------------------
Dim n As Long
Dim i As Long
Dim Tps As Variant

For n = LBound(TabDonnées) To UBound(TabDonnées) ' Boucle sur les données.

    For i = n + 1 To UBound(TabDonnées) ' Boucle depuis la donnée suivante.

        ' Gestion de l'ordre croissant ou décroissant.
        If (OrdreCroissant = True And TabDonnées(i) < TabDonnées(n)) _
        Or (OrdreCroissant = False And TabDonnées(i) > TabDonnées(n)) Then
            Tps = TabDonnées(n)             ' Mémorise la donnée à échanger
            TabDonnées(n) = TabDonnées(i)   ' Échange la donnée...
            TabDonnées(i) = Tps             ' Avec la donnée mémorisée
        End If

    Next i

Next n

End Function
```

Notre procédure donne satisfaction, pourtant elle comporte un défaut : elle fait de nombreux échanges, et chaque échange nécessite trois traitements. De plus la gestion de l'ordre de tri croissant ou décroissant nécessite une condition complexe. Le tout demande des ressources importantes, donc du temps. Il est possible que la fonction soit trop longue à exécuter sur des tableaux plus volumineux. Nous devons trouver un algorithme plus rapide.

Nous gardons notre boucle sur les données. Ici de N à N+4. Nous lisons toutes les données de N+1 à N+4, et mémorisons l'indice de la valeur la plus petite, ici c'est N+3. La lecture terminée, nous échangeons N avec N+3. Nous continuions notre boucle en lisant les données de N+1 à N+4, puis de N+2 à N+4. Ainsi de suite.

Les données sont classées par ordre croissant. Si l'ordre décroissant est désiré, il faut échanger les données.

Ce qui donne la représentation suivante :



```
'-------------------------------------------------------------------------------
Sub TriDonnéesBis(ByRef TabDonnées() As Variant, _
                  Optional OrdreCroissant As Boolean = True)
 '-------------------------------------------------------------------------------
Dim i As Long
Dim n As Long
Dim RefPos As Long
Dim Tps As Variant

For n = LBound(TabDonnées) To UBound(TabDonnées) ' Boucle sur tous les éléments.
    RefPos = n ' Indice de référence = élément actuel
    For i = n + 1 To UBound(TabDonnées) ' Boucle sur les éléments suivants.
        ' Si un élément est plus petit que l'élément de référence alors il devient
        ' l'élément de référence (nous mémorisons son indice).
        If TabDonnées(i) < TabDonnées(RefPos) Then RefPos = i
    Next i ' RefPos contient l'indice du plus petit élément.
    If RefPos <> n Then ' Si un élément plus petit que l'élément actuel est trouvé.
        Tps = TabDonnées(n) ' alors fait l'échange.
        TabDonnées(n) = TabDonnées(RefPos)
        TabDonnées(RefPos) = Tps
    End If
Next n

If OrdreCroissant = False Then ' Si Ordre décroissant alors échange les données.
    For n = LBound(TabDonnées) To UBound(TabDonnées)
        Tps = TabDonnées(UBound(TabDonnées) - n)
        TabDonnées(UBound(TabDonnées) - n) = TabDonnées(n)
        TabDonnées(n) = Tps
        If UBound(TabDonnées) - n < n Then Exit For
    Next n
End If

End Sub
```

Vous trouverez sur internet l'algorithme QuickSort, qui comme son nom l'indique, permet de trier rapidement les données. Wikipédia et d'autres sites donnent des informations plus précises sur la logique mise en œuvre.

Il existe plusieurs versions de cet algorithme, attention, toutes ne se valent pas. J'ai repris celle qui est la plus rapide. Vous pouvez remplacer la déclaration **Variant** par **Long** pour optimiser l'algorithme pour le traitement des entiers.

Sources : **http://rosettacode.org/wiki/Sorting_algorithms/Quicksort**

```
'-------------------------------------------------------------------------------
Private Sub QuickSort(ByRef TabDonnées() As Variant, ByVal Gauche As Long, _
                      ByVal Droite As Long)
'-------------------------------------------------------------------------------
Dim i As Long, j As Long, Temp As Variant, Pivot As Variant

i = Gauche
j = Droite
Pivot = TabDonnées((Gauche + Droite) / 2)

Do
    While Pivot > TabDonnées(i): i = i + 1: Wend
    While TabDonnées(j) > Pivot: j = j - 1: Wend

    If j + 1 > i Then ' If i <= j Then
        Temp = TabDonnées(i)
        TabDonnées(i) = TabDonnées(j)
        TabDonnées(j) = Temp
        j = j - 1: i = i + 1
    End If

Loop Until i > j ' ou Loop While i < j

If Gauche < j Then Call QuickSort(TabDonnées(), Gauche, j)
If i < Droite Then Call QuickSort(TabDonnées(), i, Droite)
End Sub
```

Il ne reste plus qu'à comparer nos trois algorithmes, sur un tableau volumineux.

Pour cela j'utilise la fonction EXCEL « **=ALEA.ENTRE.BORNES(0;10000)** » que je recopie sur 10 000 lignes et que je colle en valeurs.

Pour chronométrer la durée des traitements, dans la procédure « ChargeLesDonnées » je déclare une variable T de type **Double**. Avant l'appel d'une fonction de tri, je mémorise l'heure : **T = Timer**

Après le traitement, j'affiche la durée écoulée : **Timer - T**

Résultat de la course entre ces trois traitements :

- TriDonnées(MonTableau(), True) dure 14,3 secondes.
- TriDonnéesBis(MonTableau(), True) dure 2,24 secondes. Effectivement, cela valait la peine de rechercher un algorithme moins gourmand en ressources.
- QuickSort(MonTableau(), **LBound**(MonTableau), **UBound**(MonTableau)) : 0,02 seconde. Il n'y a pas photo. Cet algorithme est 100 fois plus rapide que notre meilleur algorithme.

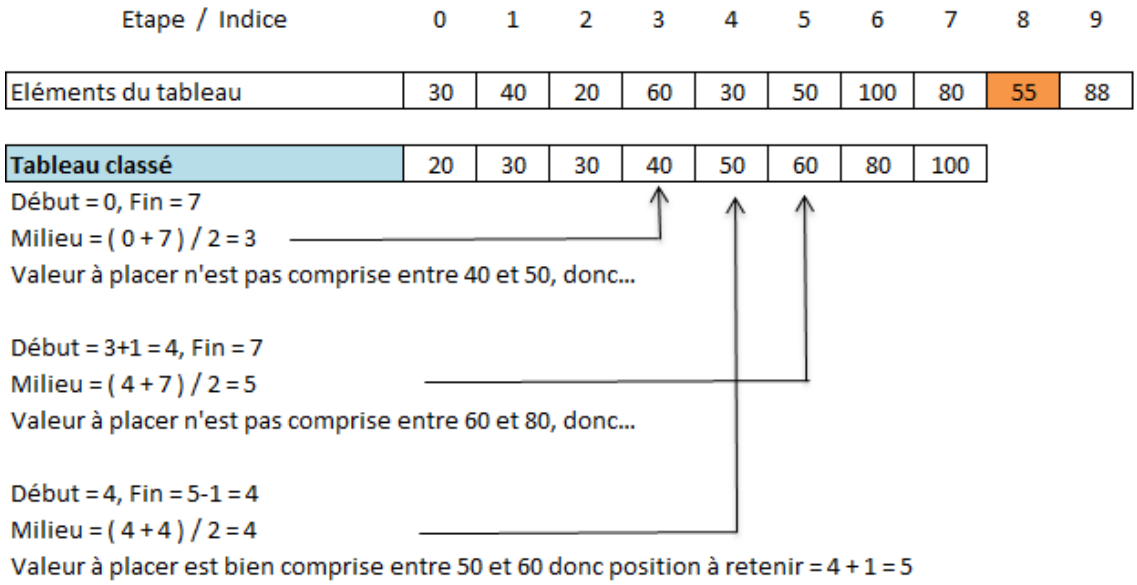
Il faudrait au moins, pour sauver l'honneur, trouver un algorithme qui permet de passer sous la barre symbolique de la seconde.

Pour gagner en temps de traitement, il faut moins de tests conditionnels. Ce sont eux qui sont gourmands en ressources.

Armé d'une feuille et d'un crayon, après deux jours de réflexion, j'ai un nouvel algorithme en tête…

Je pars sur un principe de deux tableaux. Les éléments du tableau source vont être lus par une boucle. L'élément analysé sera classé immédiatement dans le second tableau. Pour trouver son emplacement dans ce tableau classé par ordre croissant, il est trop long de lire un à un les éléments déjà classés. Une recherche par dichotomie est bien plus rapide. Une fois connu l'emplacement où insérer l'élément, il faut déplacer vers la fin du tableau les éléments déjà classés afin de libérer la place, puis y copier l'élément analysé.

Par exemple, pour classer l'élément de valeur 55 dans un tableau déjà classé : 20, 30, 30, 40, 50, 60, 80, 100, les étapes à suivre sont les suivantes :



L'algorithme de recherche dichotomique (non optimisé, pour une meilleure compréhension), est décrit dans la fonction « TableauRecherchePosition » à la page suivante.

Notez que lorsque l'on doit insérer un nouvel élément dans la partie basse du tableau des données déjà classées, plutôt que de décaler les données vers la fin du tableau, ce qui déplace de nombreuses données, il est plus judicieux de fonctionner à l'envers, en déplaçant les données vers l'autre côté, quitte à utiliser des indices négatifs, ce que le VBA permet.

```
'-------------------------------------------------------------------------------
Private Sub TableauDécalerElément(ByRef TabDonnées() As Variant, _
        ByRef IndiceElément As Long, ByRef IndiceMini As Long, ByRef IndiceMaxi As Long)
'-------------------------------------------------------------------------------
Dim i As Long

If IndiceElément > (IndiceMaxi + IndiceMini) / 2 Then
    For i = IndiceMaxi To IndiceElément Step -1 ' Décale vers le haut.
        TabDonnées(i + 1) = TabDonnées(i)
    Next i
    IndiceMaxi = IndiceMaxi + 1 ' Nouvel indice maxi du tableau.
Else
    For i = IndiceMini To IndiceElément         ' Décale vers le bas.
        TabDonnées(i - 1) = TabDonnées(i)
    Next i
    IndiceMini = IndiceMini - 1 ' Nouvel indice mini du tableau.
    IndiceElément = IndiceElément - 1 ' Décale l'emplacement du nouvel élément.
End If

End Sub
```

Le code complet de la fonction de tri :

```
'-------------------------------------------------------------------------------
Sub TriRapide(ByRef TabDonnées() As Variant)
'-------------------------------------------------------------------------------
Dim i As Long, n As Long, TabClassés() As Variant
ReDim TabClassés(-UBound(TabDonnées) - 1 To UBound(TabDonnées) + 1)
Dim TabDébut As Long, TabFin As Long

' Si moins de deux données à trier alors quitte.
If Abs(UBound(TabDonnées) - LBound(TabDonnées)) < 1 Then Exit Sub

' Classe les 2 premiers éléments par ordre croissant.
n = LBound(TabDonnées)
If TabDonnées(n + 1) > TabDonnées(n) Then i = 1
TabClassés(n) = TabDonnées(n + 1 - i)
TabClassés(n + 1) = TabDonnées(n + i)

TabDébut = LBound(TabDonnées): TabFin = LBound(TabDonnées) + 2 ' Limites tableau classé

' Boucle sur les autres éléments à classer.
For n = 2 + LBound(TabDonnées) To UBound(TabDonnées)
    ' Recherche la position dans la liste des éléments classés.
    i = TableauRecherchePosition(TabClassés(), TabDébut, TabFin, TabDonnées(n))
    ' Décale les éléments déjà classés pour faire une place.
    Call TableauDécalerElément(TabClassés(), i, TabDébut, TabFin)
    ' Insère l'élément dans la liste des éléments classés.
    TabClassés(i) = TabDonnées(n)
Next n

' Retourne le tableau classé :
n = LBound(TabDonnées)
For i = TabDébut To TabFin - 1
    TabDonnées(n) = TabClassés(i):
    n = n + 1
Next i
End Sub

'-------------------------------------------------------------------------------
```

```
Function TableauRecherchePosition(ByRef TabDonnées() As Variant, ByVal Début As Long, _
                                  ByVal Fin As Long, ByVal ValRecherchée As Variant) As Long
'------------------------------------------------------------------------------
Dim Milieu As Long

' Si nouvelle extrémité inférieure ou supérieure
If ValRecherchée <= TabDonnées(Début) Then TableauRecherchePosition = Début: Exit Function
If ValRecherchée >= TabDonnées(Fin - 1) Then TableauRecherchePosition = Fin: Exit Function

Do
    Milieu = (Début + Fin) / 2 ' Calcule le milieu du tableau borné par Début et Fin.
    ' Si l'élément à classer est compris entre Milieu et Milieu+1
    If ValRecherchée >= TabDonnées(Milieu) And ValRecherchée <= TabDonnées(Milieu + 1) Then
        TableauRecherchePosition = Milieu + 1 ' Retourne la position où insérer l'élément.
        Exit Do                               ' Sort de la boucle
    End If
    If ValRecherchée > TabDonnées(Milieu) Then  ' Compare l'élément avec le milieu
        Début = Milieu + 1                      ' Nouvelle borne de début.
    Else
        Fin = Milieu - 1                        ' Nouvelle borne de fin.
    End If
Loop
End Function
```

Testons la procédure : 0,37 seconde. C'est 20 fois plus que QuickSort, mais 50 fois moins que notre première procédure.

Grâce à ces exercices, vous manipulez désormais les tableaux de données avec aisance, ainsi que la recherche par dichotomie, une technique qui permet de réduire considérablement le temps d'exécution de certains traitements. Voici ici une version plus perfectionnée que la procédure présentée à la page précédente, qui gère le cas où la valeur cherchée n'est pas trouvée (retourne l'indice où devrait être incluse la valeur et renseigne Trouvé = Faux), ainsi que le cas où plusieurs éléments ont la valeur cherchée (retourne l'indice de la première ou de la dernière valeur et renseigne Trouvé = Vrai) :

```
'------------------------------------------------------------------------------
Public Function RechercheDichotomique(TabDonnées() As Variant, ByVal ValRecherchée As Variant, _
                                      Optional PremierSiEgalité As Boolean = True, _
                                      Optional ByRef Trouvé As Boolean) As Long
'------------------------------------------------------------------------------
' Retourne suivant que ValRecherchée est trouvée ou non :
' - Si trouvée : l'indice de la première valeur (si PremierSiEgalité=Vrai) et renseigne
 Trouvé=Vrai.
'                l'indice de la dernière valeur (si PremierSiEgalité=Faux) et renseigne
 Trouvé=Vrai.
' - Si non trouvée : l'indice où devrait être incluse la valeur et renseigne Trouvé=Faux.
'------------------------------------------------------------------------------
Dim Début As Long, Fin As Long, i As Long, Anc As Long

Trouvé = False
Début = LBound(TabDonnées): Fin = UBound(TabDonnées): Anc = Début

If ValRecherchée > TabDonnées(Fin) Then RechercheDichotomique = Fin + 1: Exit Function
If ValRecherchée < TabDonnées(Début) Then RechercheDichotomique = Début: Exit Function

' Méthode 1: En cas d'égalité dans les valeurs trouvées alors c'est l'indice le plus bas :
If PremierSiEgalité = True Then
    Do
        i = (Début + Fin) / 2
        If ValRecherchée > TabDonnées(i) Then Début = i + 1: Anc = Début Else Fin = i - 1

    Loop While Début <= Fin

    If ValRecherchée = TabDonnées(Anc) Then Trouvé = True
    RechercheDichotomique = Anc
    Exit Function
End If

' Méthode 2: En cas d'égalité dans les valeurs trouvées alors retourne l'indice le plus haut :
Do
```

```
    i = (Début + Fin) / 2
    If ValRecherchée >= TabDonnées(i) Then Début = i + 1: Anc = i Else Fin = i - 1

Loop While Début <= Fin

If ValRecherchée = TabDonnées(Anc) Then Trouvé = True Else Anc = Anc + 1
RechercheDichotomique = Anc

End Function
```

VIII - Regrouper ses procédures génériques dans un module

Certaines des procédures que nous avons étudiées sont génériques, c'est-à-dire qu'elles peuvent être réutilisées telles quelles. C'est par exemple le cas des procédures « TriRapide » ou « RechercheDichotomique ». Elles pourront nous être utiles dans d'autres projets, alors pour les retrouver facilement nous allons les regrouper dans une « boîte à outils », qui nous suivra dans toutes nos applications et sera complétée au fur et à mesure de nos progrès en programmation.

À partir de l'éditeur VBA, choisir dans le menu « Insertion », « Module ».

Copiez en en-tête du module, les deux instructions suivantes pour respectivement, forcer la déclaration des variables, indiquer à EXCEL que pour ses comparaisons de texte, la lettre « A » équivaut à la lettre « a », sans cette instruction EXCEL va classer les majuscules avant les minuscules et « B » sera classé avant « a » :

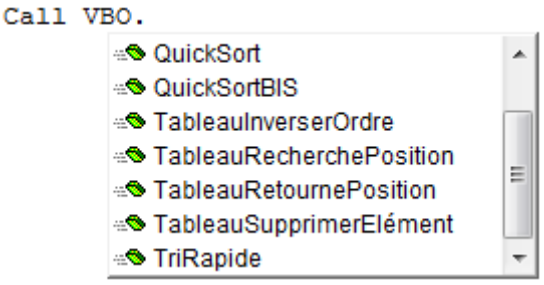
```
Option Explicit
Option Compare Text
```

Puis copiez les procédures « TriRapide », « RechercheDichotomique », « QuickSort », ainsi que les autres procédures que vous souhaitez avoir dans votre boîte à outils.
Renommer le module créé (touche [F4], après avoir sélectionné le module) en « VBO ». Votre boîte à outils est créée.

Par usage (c'est facultatif) nous ferons précéder le nom des procédures du nom du module « VBO ».

Ainsi l'appel à la fonction « TriRapide » devient : `Call VBO.TriRapide(MonTableau())`

Lorsque vous tapez « VBO. » la liste des procédures publiques du module s'ouvre…



Le fait de préciser le nom du module contenant la procédure permet aussi d'utiliser des procédures qui portent le même nom, mais situées dans des modules différents.

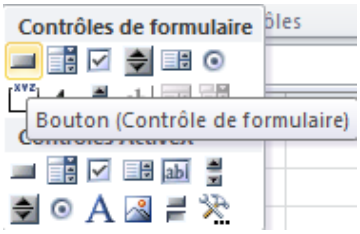
Notre application ne comporte désormais plus qu'une seule procédure spécifique au classeur EXCEL actif, et qui ne pourra probablement pas être réutilisée telle quelle dans une autre application, alors que les procédures généralistes que nous pouvons réemployer sont codées dans un module à part.

```
'-------------------------------------------------------------------------------
Sub ChargeLesDonnées()
'-------------------------------------------------------------------------------
Dim MonTableau() As Variant  ' Déclare un tableau Variant dynamique.
```

```
Dim y As Long                ' Indique la ligne à analyser.
Dim i As Long                ' Indice du tableau.
Dim T As Double              ' Pour calculer la durée du traitement.

y = 1
While Cells(y, 1) <> ""                 ' Boucle sur les lignes de la colonne A.
    ReDim Preserve MonTableau(i)        ' Redimensionne le tableau d'après l'indice.
    MonTableau(i) = Cells(y, 1)         ' Mémorise la valeur de la cellule.
    i = i + 1                           ' Incrémente l'indice du tableau.
    y = y + 1                           ' Passe à la ligne suivante.
Wend

T = Timer ' Mémorise l'heure avant le lancement du traitement.

Call VBO.TriRapideBis(MonTableau(), 0)      ' Trie les données (ordre croissant).

Durée = Timer - T                       ' Calcule la durée du traitement.

Application.ScreenUpdating = False              ' Bloque la mise à jour écran.
For y = LBound(MonTableau) To UBound(MonTableau)  ' Boucle sur les éléments.
    Cells(y + 1, 2) = MonTableau(y)             ' Affiche l'élément en B
Next y
Application.ScreenUpdating = True               ' Mise à jour de l'écran.
MsgBox "Durée =  " & Durée                      ' Affiche la durée du traitement.

End Sub
```

Jusqu'à présent, la procédure était exécutée depuis l'éditeur VBA. Nous allons créer un « bouton » sur la feuille de calcul d'EXCEL pour lancer cette procédure.

Dans le menu « Développeur », choisissez « Insérer » et cliquez sur le « bouton » :



Lorsque la souris passe sur la feuille, le curseur prend la forme d'un plus.

Cliquez sur la feuille et maintenez le clic gauche enfoncé pour définir la taille du bouton, puis relâchez le clic.

Dans la liste déroulante qui s'ouvre, piochez la procédure à associer au bouton.

Un clic droit permet de modifier le libellé du bouton.

Vous venez de créer un événement. À chaque clic sur le bouton, la procédure est lancée.

Il existe d'autres types d'événements…

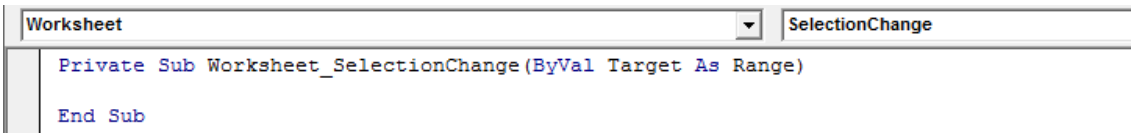
IX - Les événements

Nous avons codé nos procédures dans plusieurs modules. Et pour les exécuter nous devons soit les lancer manuellement depuis l'éditeur VBA, soit les lier à un bouton. Une procédure peut aussi être codée dans une feuille du classeur, ou, dans le classeur. Dans ce cas, la procédure codée sera lancée sur un événement lié à la feuille, ou au classeur.

Par exemple, sur la feuille 4 du classeur, pour rendre la cellule « B3 » inaccessible, j'utilise l'événement déclenché « Sur changement de cellule sélectionnée » et je code une procédure qui contrôle si la cellule active (passée en argument Target) est « B3 ». Dans ce cas je sélectionne la cellule de la ligne suivante :

Cells(y+1,x).Activate avec y = **Target.Row** et x = **Target.column**

Faire un clic droit sur l'onglet de la feuille 4, et choisir dans la liste déroulante « Visualiser le code ».
L'éditeur VBA s'ouvre. Piochez dans la liste de gauche « Worksheet » et dans la liste de droite « SelectionChange ».
Une procédure vierge s'ouvre :

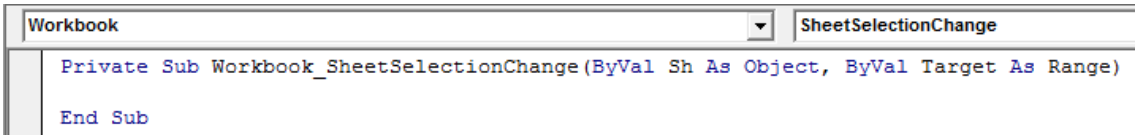


Écrire le code suivant, qui sera exécuté chaque fois que la cellule active de la feuille changera :

```
Private Sub WorkSheet_SelectionChange(ByVal Target As Range)
' Événement déclenché à chaque changement de cellule sélectionnée.
' L'argument Target représente la cellule active.

If Target.Row = 3 And Target.Column = 2 Then ' Si cellule sélectionnée = B3
    Cells(Target.Row + 1, Target.Column).Activate 'Active la cellule suivante.
End If

End Sub
```

Notez que dans le code, nous pouvons aussi faire appel à une procédure d'un module (sauf si elle a été déclarée **Private)**. C'est une autre façon d'appeler nos procédures.

L'événement est lié à la feuille 4, la feuille 3 n'est donc pas concernée par cette procédure et sa cellule « B3 » peut être sélectionnée.
Pour que cette procédure concerne toutes les feuilles du classeur, il faut que l'événement soit déclaré au niveau du classeur et non plus de la feuille.
Dans la partie droite de l'éditeur VBA, cliquez sur « ThisWorkbook ». Piochez dans la liste de gauche de l'éditeur « Workbook » et dans la liste de droite « SheetSelectionChange ». Une procédure vierge s'ouvre :



Écrire le code suivant, qui sera exécuté chaque fois que la cellule active changera dans le classeur :

```
Private Sub WorkBook_SheetSelectionChange(ByVal Sh As Object, ByVal Target As Array)
' Événement déclenché à chaque changement de cellule sélectionnée.
' L'argument Sh représente la feuille active.
' L'argument Target représente la cellule active.

If Target.Row = 3 And Target.Column = 2 Then ' Si cellule sélectionnée = B3
    Cells(Target.Row + 1, Target.Column).Activate 'Active la cellule suivante.
End If

End Sub
```

On retrouve ici le principe de la portée des procédures **Private** et **Public**.

- Les événements d'une feuille ne concernent que la feuille en question.
- Les événements du classeur, concernent toutes les feuilles du classeur.

Certains événements sont spécifiques au classeur et n'existent pas pour les feuilles.

Autre façon de lancer une procédure, lui associer une touche du clavier avec :

```
Application.OnKey "Touche", "Procédure"
```

- Touche : est le code la touche concernée ;
- Procédure : est le nom de la procédure à exécuter. Ou vide pour annuler le lien entre la touche et la procédure.

Utilisez cette technique avec modération, car si le code de la procédure plante entre l'attribution du raccourci et sa libération, le lien associé à la touche du clavier reste actif.

Par exemple, pour lancer la procédure « ChargeLesDonnées » par la touche « F5 » lorsque l'utilisateur est sur la feuille 4 : il faut activer ce lien quand l'utilisateur active la feuille 4 (événement « Activate » de la feuille) , et désactiver le lien quand l'utilisateur quitte la feuille 4 (événement « Deactivate » de la feuille).

Ce qui donne le code suivant :

```
Private Sub WorkSheet_Activate()
' Événement déclenché chaque fois que l'on active la feuille

Application.OnKey "{F5}", "ChargeLesDonnées" ' Activer le lien de la touche "F5"

End Sub

Private Sub WorkSheet_Deactivate()
' Événement déclenché chaque fois que l'on quitte feuille

Application.OnKey "{F5}", "" ' Désactiver le lien avec la touche "F5"

End Sub
```

Une procédure peut aussi être exécutée à un moment précis, par exemple à 17 heures :

```
Application.OnTime TimeValue("17:00:00"), "ChargeLesDonnées"
```

Extrait de la documentation EXCEL des principaux événements liés à une feuille :

Nom	Description
Activate	Se produit lorsqu'une feuille de calcul est activée.
BeforeDoubleClick	Se produit lorsqu'un utilisateur double-clique sur une feuille de calcul, avant l'action de double-clic par défaut.
BeforeRightClick	Se produit lorsqu'un utilisateur clique avec le bouton droit de la souris sur une feuille de calcul, avant l'action par défaut de clic avec le bouton droit.
Calculate	Se produit après le recalcul de la feuille de calcul, pour l'objet Worksheet.
Change	Se produit lorsque les cellules de la feuille de calcul sont modifiées par l'utilisateur ou par un lien externe.
Deactivate	Se produit lorsque la feuille de calcul est désactivée.
SelectionChange	Se produit lorsque la sélection change dans une feuille de calcul.

Extrait de la documentation EXCEL des principaux événements liés au classeur :

Nom	Description
Activate	Se produit lorsqu'un classeur est activé.
AfterSave	Se produit après l'enregistrement du classeur.
BeforeClose	Se produit avant la fermeture du classeur. Si le classeur a été modifié, cet événement se produit avant que l'utilisateur soit invité à enregistrer ses modifications.
BeforePrint	Se produit avant l'impression du classeur (ou de tout élément de celui-ci).
BeforeSave	Se produit avant l'enregistrement du classeur.
Open	Se produit lorsque le classeur est ouvert.
SheetActivate	Se produit lorsqu'une feuille est activée.
SheetChange	Se produit lorsque des cellules d'une feuille de calcul sont modifiées par l'utilisateur ou par un lien externe.
SheetDeactivate	Se produit lorsqu'une feuille de calcul est désactivée.

Comme vous pouvez le constater, les événements permettent de couvrir une large gamme de besoins.

Par exemple, nous pouvons archiver sur la feuille 5 les modifications faites dans la feuille 4, en indiquant les coordonnées de la cellule modifiée, son ancienne valeur, la date de modification, l'utilisateur concerné, la nouvelle valeur. Pour éviter les débordements, nous n'archivons que les 20 000 dernières modifications.

Nous allons utiliser dans un premier temps l'événement « Worksheet_SelectionChange » pour mémoriser dans une variable publique la valeur de la cellule sélectionnée, c'est-à-dire avant sa modification, puis dans un second temps, l'événement « Worksheet_Change » activé lors de la modification de la valeur de la cellule.

Ce qui donne le code suivant sur la feuille 4 :

```
Dim AncienneValeur As Variant

Private Sub Worksheet_SelectionChange(ByVal Target As Range)
' Evénement déclenché à chaque changement de cellule sélectionnée.
' L'argument Target représente la cellule active.

AncienneValeur = Target.Value ' Mémorise la valeur de la cellule avant modification

End Sub

Private Sub Worksheet_Change(ByVal Target As Range)
' Evénement déclenché quand la valeur de la cellule change.
' L'argument Target représente la cellule active.

Sheets("Feuil5").Rows(2).Insert ' Insertion d'une nouvelle ligne en ligne 2
Sheets("Feuil5").Cells(2, 1) = Target.Address ' Coordonnées de la cellule
Sheets("Feuil5").Cells(2, 2) = AncienneValeur ' Ancienne valeur
Sheets("Feuil5").Cells(2, 3) = Now            ' jour et heure de modification
Sheets("Feuil5").Cells(2, 4) = Application.UserName ' Utilisateur
Sheets("Feuil5").Cells(2, 5) = Target.Value         ' Nouvelle valeur

Sheets("Feuil5").Rows(20000).Delete   ' Suppression de la ligne 20 000

End Sub
```

Grâce à l'événement « *BeforeClose* » déclenché avant de fermer le classeur, nous allons envoyer un message via OUTLOOK à l'administrateur lorsque l'utilisateur quitte l'application, en lui indiquant les modifications faites. Nous voulons supprimer ce message du dossier « boîte d'envoi » de l'utilisateur pour ne pas l'encombrer de données inutiles…

Mais avant nous allons ajouter une nouvelle procédure dans notre boîte à outils.
La fonction « EnvoiMailOutLook » comporte plusieurs arguments :

- StrSendTo : est l'adresse de messagerie du destinataire. Nous prendrons celle de l'administrateur, préalablement renseignée en cellule « A1 » de la feuille 5.
- StrSubject : est l'objet du message. Ici nous indiquerons le nombre de modifications faites, et l'utilisateur concerné.
- StrBodyText : est le corps du message. C'est le détail de ces modifications, cellule concernée, valeur avant et après intervention.
- StrAttachment (facultatif) : est l'éventuel fichier à mettre en pièce jointe. Pour l'instant nous n'utiliserons pas cet argument.
- SupprimerBoiteEnvoi (facultatif) : indique s'il faut supprimer le message envoyé de la boîte des messages envoyés. Par défaut cet argument est à « Faux ».

Nous allons aussi gérer les éventuelles erreurs qui pourraient survenir dans le traitement.
Avant d'étudier le code de la procédure, nous allons donc faire une brève parenthèse pour évoquer la gestion des erreurs…

X - La gestion des erreurs

Par défaut en VBA, lorsqu'une erreur se produit dans un traitement, un message s'affiche, parfois explicite, souvent très avare d'explication, et surtout, le traitement plante, laissant l'utilisateur dans une impasse.

Bien sûr, comme tout bon langage de programmation, VBA permet d'intercepter une erreur de traitement, et de proposer une alternative au plantage fatidique. Cependant, vous constaterez que la gestion des erreurs est souvent négligée par les développeurs, car elle est très chronophage : il faut en effet penser à tous les cas de figure qui peuvent faire planter un traitement, parfois banal, et écrire un code en conséquence, qui peut se révéler plus long que le code principal.

VBA met quatre mots clés à disposition des développeurs pour gérer les erreurs :

- On Error Resume Next : indique qu'en cas d'erreur, le traitement doit continuer comme si de rien n'était et passer à la ligne suivante. Généralement ça ne règle pas le problème, mais c'est très rapide à mettre en œuvre et ça peut rendre service ;
- On Error Goto NomEtiquette : indique que le traitement doit se brancher à l'étiquette indiquée. En principe l'étiquette, déclarée par « NomEtiquette: » contient un code qui permet de gérer réellement l'erreur. Le nom de l'étiquette est suivi du signe deux-points.

  Il est possible de sortir de l'étiquette et de retourner à la ligne qui a généré l'erreur par le mot clé **Resume** ou de se connecter à une autre étiquette par le mot clé Resume NomEtiquette ;
- On Error Goto 0 : indique que la gestion personnalisée des erreurs est terminée. VBA reprend la main et en cas d'erreur, le traitement plante ;
- **Err** : cet objet contient les informations sur l'erreur (numéro de l'erreur, description, source…)

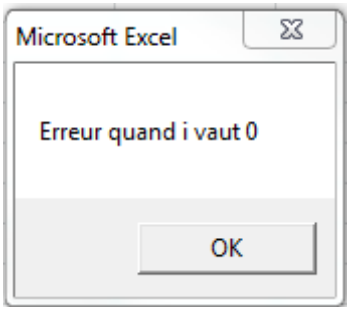
Pour illustrer ces propos, je provoque volontairement une erreur dans le code suivant en appelant la cellule (0,1) qui n'existe pas. Grâce à On Error Resume Next le traitement se poursuit naturellement :

```
Sub TestErreur()
Dim i As Integer, MaValeur As Variant
```

```
On Error Resume Next ' Ignore les erreurs
For i = 0 To 9
    MaValeur = Cells(i, 1)
Next i
End Sub
```

On Error Goto Err_Test permet en cas d'erreur de brancher le traitement à une étiquette pour afficher un message personnalisé, et Resume Next permet ensuite de poursuivre à la ligne suivante.

```
Sub TestErreur()
Dim i As Integer, MaValeur As Variant

On Error GoTo Err_Test ' Gestion des erreurs
For i = 0 To 9
    MaValeur = Cells(i, 1)
Next i

Err_Test: ' Étiquette de branchement en cas d'erreur
          ' ou suite du traitement ordinaire.
If Err.Number <> 0 Then    ' S'il y a une erreur (l'Objet Err contient les informations)...
    MsgBox "Erreur quand i vaut " & i    ' Mon message d'erreur,
    Resume Next                          ' Reprend à la ligne suivante,
End If                                   ' Fin du traitement de l'erreur.

End Sub
```



XI - Envoyer un message OUTLOOK depuis EXCEL

EXCEL ne parle pas OUTLOOK. Le VBA d'EXCEL va devoir apprendre les fonctions propres à OUTLOOK, les structures des objets, les énumérations et autres constantes. Cela se fait en chargeant le dictionnaire OUTLOOK, depuis l'éditeur VBA, dans le menu « Outils », « Références », en cochant « Microsoft Outlook 14.0 Object Library ».

Nous pouvons maintenant créer dans EXCEL un objet, olApp, qui représente l'application OUTLOOK :

```
Dim olApp As Outlook.Application
Set olApp = GetObject("", "Outlook.Application")
```

Puis nous créons un objet, objMail, qui représente la structure d'un message dans l'application OUTLOOK :

```
Dim objMail As Outlook.MailItem
Set objMail = olApp.CreateItem(olMailItem)
```

Nous pouvons maintenant renseigner les propriétés de cet objet vierge, à savoir le destinataire principal (objMail.To), la copie (objMail.CC), la copie cachée (objMail.BCC), le sujet (objMail.Subject), le corps (objMail.Body), le style (objMail.BodyFormat), voire ajouter une pièce jointe (objMail.Attachments.Add).

Dans notre procédure « EnvoiMailOutLook » (voir page suivante) l'argument StrAttachment, qui renseigne la pièce jointe à attacher au message, est facultatif.

Attention, car dans un code, faire référence à un argument facultatif non renseigné génère une erreur. Il faut donc au préalable s'assurer de l'existence de l'argument, avec la fonction IsMissing(Argument) qui retourne « Vrai » si l'argument n'est pas renseigné ou « Faux » dans le cas contraire (n'est possible que si l'argument est de type **Variant**). Ainsi dans notre fonction, avant d'attacher un fichier en pièce jointe nous procédons à deux contrôles :

- vérification que l'argument est renseigné : If IsMissing(StrAttachment) = False Then ;
- vérification que le chemin indiqué est valide : If Dir(StrAttachment) <> "".

Alors nous pouvons attacher la pièce jointe StrAttachment : objMail.Attachments.Add StrAttachment.

Tout est fin prêt pour envoyer le message : objMail.Send.

Pour supprimer ce message de la boîte d'envoi où il s'est copié automatiquement, nous devons procéder en deux étapes.

D'abord il faut créer un objet, MyNameSpace, pour accéder à l'espace de travail de OUTLOOK :

```
Dim MyNameSpace As Outlook.Namespace
Set MyNameSpace = olApp.GetNamespace("MAPI")
```

Ensuite, parmi cet espace qui regroupe toutes les boîtes de l'application, il faut créer un objet, MaBoite, issu de cet espace, MyNameSpace, et qui représente la boîte d'envoi (olFolderSentMail) :

```
Dim MaBoite As Object
Set MaBoite = MyNameSpace.GetDefaultFolder(olFolderSentMail)
```

Ainsi, nous accédons à une propriété de l'élément i de la boîte d'envoi qui contient MaBoite.Items.Count éléments par la syntaxe suivante :

```
MaBoite.Items(i).Membre
```

Membre peut être une propriété du message, comme son sujet (Subject), ou une méthode, comme supprimer (Delete). Nous reviendrons sur ces notions dans les pages suivantes. Ici nous souhaitons supprimer le message que nous venons d'envoyer et que nous identifions parmi tous les autres messages, grâce au sujet qui correspond à celui du message envoyé.

Les éléments de la boîte d'envoi peuvent être de nature autre qu'un message, être par exemple des rendez-vous. Ainsi l'élément n'a pas forcément la propriété Subject et sa lecture déclenche une erreur. C'est là que On Error Resume Next nous rend service, en poursuivant le traitement sans faire planter la procédure.

Le code de la procédure se trouve sur la page suivante.

Vous avez remarqué, qu'avant du supprimer le message, nous attendons une seconde pour qu'il y soit :

```
T = Timer: While T+1 > Timer: Wend
```

Nous avons une gestion des erreurs dans la première partie du traitement jusqu'à l'envoi du message, qui branche le code à l'étiquette Err_EnvoiMailOutLook en cas d'erreur :

```
On Error GoTo Err_EnvoiMailOutLook
```

Et indique que le message n'a pas été envoyé, puis sort de l'application en fermant correctement les objets ouverts, en se branchant directement à l'étiquette Exit_EnvoiMailOutLook :

```
Resume Exit_EnvoiMailOutLook
```

En cas d'erreur, donc, la ligne EnvoiMailOutLook = True n'est pas lue. Et la fonction retourne **False** par défaut.

En règle générale, il est souvent très utile d'utiliser une fonction de type **Boolean** pour retourner la valeur **True** si tout s'est bien passé.
Par défaut la valeur retournée est **False**, il n'est donc pas nécessaire d'initialiser cette valeur dans le code de la fonction.

Cette valeur retournée peut être utilisée par la procédure appelante pour influencer la suite du traitement.

```
If VBO.EnvoiMailOutLook(Admin, Sujet, Message, , True) = False Then
    [Traitement]
End if
```

L'initialisation des variables avec **Dim** n'est pas faite obligatoirement en début du module. Elle peut se faire à n'importe quel moment du code, comme dans notre fonction, où Dim MyNameSpace et Dim MaBoite ne sont initialisés qu'en cas de nécessité.

Notre fonction pour envoyer un message dans OUTLOOK depuis EXCEL :

```
'-------------------------------------------------------------------------------
Function EnvoiMailOutLook(ByVal StrSendTo As Variant, ByVal StrSubject As String, _
                          ByVal StrBodyText As String, Optional ByVal StrAttachment, _
                          Optional SupprimerBoiteEnvoi As Boolean = False) As Boolean
'-------------------------------------------------------------------------------
Dim i As Long, T As Double

On Error GoTo Err_EnvoiMailOutLook ' Étiquette de branchement en cas d'erreur.

Dim olApp As Outlook.Application
Set olApp = GetObject("", "Outlook.Application") ' Déclaration objet Outlook.

Dim objMail As Outlook.MailItem
Set objMail = olApp.CreateItem(olMailItem)       ' Déclaration objet Message.

objMail.To = StrSendTo        ' Destinataire (.CC copie et .BCC copie cachée)
objMail.Subject = StrSubject  ' Sujet
objMail.Body = StrBodyText    ' Texte
objMail.BodyFormat = olFormatRichText  ' Format du message RTF (texte enrichi)

' Ajout d'une pièce jointe si passée en argument, après contrôle de son chemin.
If IsMissing(StrAttachment) = False Then
    If Dir(StrAttachment) <> "" Then objMail.Attachments.Add StrAttachment
End If

objMail.Send              ' Envoie le message
EnvoiMailOutLook = True   ' Retourne True si tout s'est bien passé

If SupprimerBoiteEnvoi = True Then ' S'il faut le supprimer de la boite d'envoi.

    T = Timer: While T + 1 > Timer: Wend ' Attente 1 seconde l'envoi du message.
    On Error Resume Next                 ' Ignore les éventuelles erreurs.

    Dim MyNameSpace As Outlook.Namespace
    Set MyNameSpace = olApp.GetNamespace("MAPI")

    Dim MaBoite As Object
    Set MaBoite = MyNameSpace.GetDefaultFolder(olFolderSentMail)

    For i = 1 To MaBoite.Items.Count ' Boucle sur les messages de la boite d'envoi
        If MaBoite.Items(i).Subject = StrSubject Then ' Si c'est ce message
            MaBoite.Items(i).Delete                   ' alors le supprimer.
            Exit For                                  ' Sort de la boucle.
        End If
    Next i
```

```
End If

Err_EnvoiMailOutLook:       ' Étiquette branchée en cas d'erreur
If Err.Number <> 0 Then     ' Si erreur, alors Description de l'erreur.
   MsgBox "Message non envoyé suite erreur : " & Err.Description, vbCritical
   Resume Exit_EnvoiMailOutLook
End If

Exit_EnvoiMailOutLook:      ' Étiquette de fin de traitement.
    Set olApp = Nothing     ' Suppression des objets en mémoire.
    Set objMail = Nothing   ' Suppression des objets en mémoire.
    Set MaBoite = Nothing   ' Suppression des objets en mémoire.

End Function
```

Cette procédure sera déclenchée par l'événement « *BeforeClose* », lorsque l'utilisateur quitte l'application.

Dans la partie droite de l'éditeur VBA, cliquez sur « ThisWorkbook ». Piochez dans la liste de gauche de l'éditeur « Workbook » et dans la liste de droite « BeforeClose ».

Copiez le code suivant :

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)

Dim y As Long, NbModif As Integer
Dim Message As String, Sujet As String, S As String, Admin As String

y = 2 ' Boucle sur les modifications faites
While Sheets("feuil5").Cells(y, 1) <> ""

    ' Si modification faite aujourd'hui ET par l'utilisateur actuel.
    If DateDiff("d", Sheets("feuil5").Cells(y, 3), Now) = 0 _
    And Sheets("feuil5").Cells(y, 4) = Application.UserName Then

        NbModif = NbModif + 1 ' Compte le nombre de modifications

        ' Génère le corps du message à envoyer.
        Message = Message & "Cellule " & Sheets("feuil5").Cells(y, 1) _
        & " modifiée de " & Sheets("feuil5").Cells(y, 2) _
        & " en " & Sheets("feuil5").Cells(y, 5) _
        & Chr(10)
    End If
    y = y + 1 ' Passe à la ligne suivante.

Wend

If NbModif = 0 Then Exit Sub ' Si aucune modification alors quitte
If NbModif > 1 Then S = "s"  ' Si plusieurs modifications ajoute d'un "s".

' Création du sujet du message.
Sujet = NbModif & " modification" & S & " effectuée" & S _
        & " par l'utilisateur " & Application.UserName & "" _
        & " le " & Date

Admin = Sheets("feuil5").Cells(1, 1) ' Adresse de l'administrateur

' Envoie un message à l'administrateur et indique si tout se passe bien
If VBO.EnvoiMailOutLook(Admin, Sujet, Message, , True) = True Then
    MsgBox "Votre administrateur a été informé de vos mises à jour"
End If

End Sub
```

VBA permet de développer des procédures complètes et autonomes, comme ici notre fonction pour envoyer un message dans OUTLOOK. Avec une telle fonction, le développeur n'a plus à se soucier de l'aspect technique ni de la gestion des erreurs, il lui suffit de renseigner les bonnes données dans les bons arguments et le tour est joué. De plus, la valeur retournée par la fonction permet d'agir sur la suite du traitement.

Idem pour la fonction que je vous propose ici, pour ouvrir une boîte de dialogue afin de sélectionner un fichier, sélectionner un répertoire, ou enregistrer un fichier. Suivant le besoin…

XII - Exemple d'une fonction complète en VBA pour ouvrir un fichier

Pour ouvrir une boîte de dialogue « Fichier/Ouvrir », VBA dispose de l'instruction **GetOpenFilename**, mais nous allons préférer l'objet **FileDialog** qui offre plus de possibilités, car il permet de gérer trois types de situations : sélectionner un fichier, sélectionner un répertoire, ou enregistrer un fichier.

La boîte de dialogue peut limiter les fichiers affichés d'après leur extension, via un filtre. La fonction prévoit de prérenseigner ce filtre pour les fichiers de type EXCEL, ACCESS, mais laisse aussi la possibilité au développeur de renseigner un filtre particulier, ou de cumuler des filtres.
Le titre de la boîte de dialogue et le bouton peuvent être personnalisés.
Le répertoire par défaut peut être déterminé.
La fonction retourne l'adresse (chemin et nom) de la sélection, cette information étant utilisée ultérieurement, ou une chaîne vide dans les autres cas.
La fonction utilise l'instruction **Select Case** qui permet des traitements conditionnels très puissants et peut remplacer avantageusement un ensemble d'instructions **If … Then**.

La syntaxe est la suivante :

```
Select Case ExpressionATester[Case Expression [Traitements]] ...[Case Else[AutresTraitements]]
End Select
```

Ne pas confondre avec **UCase**(Texte) qui met en majuscules le texte passé en argument.

Voici la fonction :

```
Function BoiteDialogue(FormatBoiteDialogue As MsoFileDialogType, ByVal StrTitre As String,
                       ByVal StrBouton As String, ByVal StrRépertoire As String,
                       OuvrirFichier As String, ParamArray ListeFiltre() As Variant) As String
```

Étudions ses arguments.

- FormatBoiteDialogue : est l'un des formats possibles de la boîte de dialogue. Ces formats sont contenus dans l'énumération MsoFileDialogType, intégrée dans VBA, que nous utilisons.
    - Utilisez msoFileDialogFilePicker ou msoFileDialogOpen pour sélectionner un fichier.
    - Utilisez msoFileDialogFolderPicker pour sélectionner un répertoire.
    - Utilisez msoFileDialogSaveAs pour ouvrir une boîte de dialogue « Enregistrer sous ».
- StrTitre : est le titre de la boîte de dialogue.
- StrBouton : est le libellé du bouton de la boîte de dialogue.
- StrRépertoire : permet d'indiquer le répertoire où s'ouvrira la boîte de dialogue. Par exemple « C:\ » Mais, si la valeur USER est passée en argument, le répertoire de l'utilisateur est choisi (Application.DefaultFilePath). Si c'est la valeur ICI qui est passée en argument, c'est le répertoire où est l'application qui est repris (ThisWorkbook.Path). Si l'argument est vide, c'est le dernier répertoire utilisé qui est repris.
- OuvrirFichier : est le nom du fichier attendu à être sélectionné. Si cet argument est renseigné et que l'utilisateur sélectionne un autre fichier, un message informera l'utilisateur de son erreur et lui demandera s'il veut recommencer sa sélection. En cas de refus, la fonction retournera une chaîne vide.
- ListeFiltre : est le type de fichiers que l'on veut sélectionner. Cet argument sert de filtre à la boîte de dialogue. Il fonctionne par couple : libellé à afficher (virgule) extensions. Les différentes extensions sont séparées par un point-virgule. Par exemple : "Fichiers Exécutables,*.exe;*.mde;*.accde"

  Pour plus de souplesse, nous alimentons automatiquement ce filtre pour les types courants, Excel, Access, et tous les fichiers. Le développeur passe alors simplement en argument "EXCEL", "ACCESS", "TOUT" en minuscules ou en majuscules.

Les filtres prédéfinis et, ou, personnels peuvent se cumuler : par exemple : "EXCEL", "ACCESS", "Fichiers Exécutables,*.exe;*.mde;*.accde", "Tout"

Cet argument est déclaré **ParamArray**. Ce qui permet de passer un nombre de valeurs non déterminé, qui seront mémorisées dans un tableau de données.

L'on consultera ListeFiltre(i) pour connaître la valeur de l'élément i.

Un argument déclaré **ParamArray** doit toujours être le dernier de la procédure, et est toujours de type **Variant**.

Quelques exemples d'appels avant de voir le code de cette fonction :

- pour forcer l'utilisateur à sélectionner le fichier « SAPI.xlsm », qui devrait être dans le répertoire « Documents » de l'utilisateur. La fonction retourne le chemin et le nom du fichier si le fichier est sélectionné :

```
VBO.BoiteDialogue (msoFileDialogFilePicker , "Sélectionnez le fichier SAPI", "Ouvrir
 SAPI", "User", "Sapi.xlsm", "Excel")
```

- pour forcer l'utilisateur à sélectionner un fichier EXCEL ou ACCESS, qui devrait être dans le répertoire de l'application. La fonction retourne le chemin et le nom du fichier s'il y a une sélection :

```
VBO.BoiteDialogue (msoFileDialogFilePicker, "Sélectionnez un fichier", "Sélectionner ce
 fichier", "ici", "", "Excel", "Access")
```

- pour demander à l'utilisateur de sélectionner un répertoire, par défaut sur « P ». La fonction retourne le chemin et le nom du répertoire s'il y a une sélection :

```
VBO.BoiteDialogue(msoFileDialogFolderPicker, "Sélectionnez un dossier", "Ce
 dossier", "P:\", "", "")
```

- pour ouvrir la boîte de dialogue enregistrer sous, le fichier par défaut étant « P:\Monfichier.xls ». La fonction retourne le chemin et le nom du fichier s'il y a une validation :

```
VBO.BoiteDialogue(msoFileDialogSaveAs, "Enregistrer sous", "Valider ce Fichier", "P:
\Monfichier.xls", "", "")
```

```
'-------------------------------------------------------------------------------
Function BoiteDialogue(FormatBoiteDialogue As MsoFileDialogType, ByVal StrTitre As String, _
                       ByVal StrBouton As String, ByVal StrRepertoire As String, _
                       OuvrirFichier As String, ParamArray ListeFiltre() As Variant) As String
'-------------------------------------------------------------------------------
Dim ChoixFichier As String, Répertoire As String
Dim i As Integer, LibFiltre() As String, ExtFiltre() As String
Dim fDialog As Office.FileDialog

On Error Resume Next
For i = LBound(ListeFiltre()) To UBound(ListeFiltre()) ' Boucle sur les filtres indiqués...

    ReDim Preserve LibFiltre(i) ' Libellé du filtre.
    ReDim Preserve ExtFiltre(i) ' Extension du filtre.

    Select Case UCase(ListeFiltre(i)) ' Choix du filtre suivant l'argument.
        Case "EXCEL": LibFiltre(i) = "Fichiers EXCEL": ExtFiltre(i) = "*.xl*"
        Case "ACCESS": LibFiltre(i) = "Fichiers ACCESS": ExtFiltre(i) = "*.mdb;*.accdb"
        Case "TOUT": LibFiltre(i) = "Tous les Fichiers": ExtFiltre(i) = "*.*"
        Case Else ' Récupère libellé avant la virgule, extension après la virgule.
        LibFiltre(i) = Left(ListeFiltre(i), InStr(1, ListeFiltre(i), ",", vbTextCompare) - 1)
        ExtFiltre(i) = Mid(ListeFiltre(i), InStr(1, ListeFiltre(i), ",", vbTextCompare) + 1)
    End Select
```

```
Next i

Select Case UCase(StrRepertoire) ' Choix du répertoire suivant l'argument.
    Case "USER": Répertoire = Application.DefaultFilePath ' Répertoire Documents.
    Case "ICI": Répertoire = ThisWorkbook.Path            ' Répertoire de l'application.
    Case Else: Répertoire = StrRepertoire                 ' Prend le répertoire indiqué.
End Select

If StrBouton = "" Then StrBouton = "Ouvrir"     ' Bouton par défaut.
If StrTitre = "" Then StrTitre = "Sélectionner" ' Titre par défaut.

Set fDialog = Application.FileDialog(FormatBoiteDialogue) ' Création d'une Boîte Dialogue.
fDialog.InitialView = msoFileDialogViewDetails ' Affichage des fichiers par détail.
fDialog.AllowMultiSelect = False      ' Multisélection non autorisée.
fDialog.Title = StrTitre              ' Titre personnalisé.
fDialog.ButtonName = StrBouton        ' Libellé personnalisé du bouton.
fDialog.InitialFileName = Répertoire  ' Répertoire par défaut.
fDialog.Filters.Clear                 ' Efface les filtres.
For i = 0 To UBound(LibFiltre())      ' Boucle sur les filtres demandés...
    fDialog.Filters.Add LibFiltre(i), ExtFiltre(i) ' Ajoute le filtre et l'extension.
Next i

MonChoix:                                   ' Étiquette.
If fDialog.Show = True Then                 ' S'il y a une sélection...
    ChoixFichier = fDialog.SelectedItems(1) ' La mémoriser.
    BoiteDialogue = ChoixFichier            ' Retournera la sélection.
Else                                        ' Sinon...
    BoiteDialogue = "": Exit Function       ' Retourne vide.
End If

If OuvrirFichier <> "" Then  ' S'il faut ouvrir le fichier sélectionné.
    ' Si le fichier sélectionné est bien celui qu'il faut ouvrir :
    If InStr(1, ChoixFichier, OuvrirFichier, vbTextCompare) > 0 Then
        BoiteDialogue = ChoixFichier          ' Retournera la sélection.
    Else ' Si le fichier sélectionné n'est pas le bon demande une nouvelle sélection.
        If MsgBox("Le fichier sélectionné n'est pas " & OuvrirFichier _
            & Chr(10) & Chr(13) & "Voulez-vous recommencer la sélection ?", _
            vbYesNo, "Erreur de fichier") = vbYes Then GoTo MonChoix
        BoiteDialogue = ""      ' Retournera vide.
    End If
End If

End Function
```

XIII - Utiliser les boîtes de dialogue intégrées d'EXCEL

Le VBA permet d'accéder aux différentes boîtes de dialogue disponibles dans EXCEL, via la propriété Application.Dialogs(XlBuiltInDialog).Show, où l'énumération XlBuiltInDialog représente le type de boîte de dialogue à lancer. Recherchez dans l'aide « Énumération XlBuiltInDialog » pour obtenir une liste des boîtes de dialogue disponibles.

L'exemple suivant lance la boîte de dialogue « Fichier/Ouvrir», et ouvre le fichier sélectionné par l'utilisateur. La méthode retourne **False** si l'utilisateur a cliqué sur « Annuler », ou **True** si un fichier a effectivement été ouvert, mais ne retourne pas le nom du fichier. Et contrairement à l'instruction **GetOpenFilename**, il n'est pas possible de vérifier que le fichier sélectionné est celui attendu, avant son ouverture, ce qui peut poser un problème.

```
'-------------------------------------------------------------------------------
Sub OuvrirUnFichier()
'-------------------------------------------------------------------------------
Dim Action As Boolean

Action = Application.Dialogs(xlDialogOpen).Show ' Boîte de dialogue Ouvrir
If Action = False Then Exit Sub                 ' Si annulation, alors quitter

' ... Votre code

End Sub
```

En consultant l'aide d'EXCEL, nous apprenons que cette boîte supporte les 14 arguments suivants :

> « file_text, update_links, read_only, format, prot_pwd, write_res_pwd, ignore_rorec, file_origin, custom_delimit, add_logical, editable, file_access, notify_logical, converter »

Par contre, l'aide d'EXCEL ne donne pas plus d'explication sur l'usage de ces arguments, ni d'exemple. Il faut donc procéder par déduction où se rabattre sur internet. Ici, le premier argument, file_text, permet de prérenseigner la boîte de dialogue afin, par exemple, de sélectionner un fichier EXCEL dont le nom commence par SAPI : Application.Dialogs(xlDialogOpen).Show("SAPI*.xl*")

Attention, tous les arguments doivent être placés entre guillemets, même les valeurs de type numérique.

Exemple d'une fonction qui ouvre la boîte de dialogue intégrée « Motifs » pour sélectionner une couleur, et retourne le numéro de la couleur choisie par l'utilisateur :

```
'-------------------------------------------------------------------------------
Function ChoisirUneCouleur () As Long
'-------------------------------------------------------------------------------
' Mémorise la couleur de fond de la cellule active :
MémoriseCouleur = ActiveCell.Interior.Color
' Ouvre la Boîte de dialogue pour choisir une couleur :
Application.Dialogs(xlDialogPatterns).Show
' Renseigne la couleur choisie :
ChoisirUneCouleur = ActiveCell.Interior.Color
' Restaure la couleur d'origine à la cellule active
ActiveCell.Interior.Color = MémoriseCouleur
End Function
```

Les boîtes de dialogue sont en mode modal, c'est-à-dire que le code VBA est suspendu tant que la boîte de dialogue est ouverte. Tout comme l'instruction **Msgbox** que nous avons étudiée. Il est cependant possible de « naviguer » dans une boîte de dialogue en simulant des touches par l'instruction **Sendkeys** comme s'il s'agissait d'une saisie directe au clavier. Cette instruction doit donc être placée avant l'appel de la boîte de dialogue.

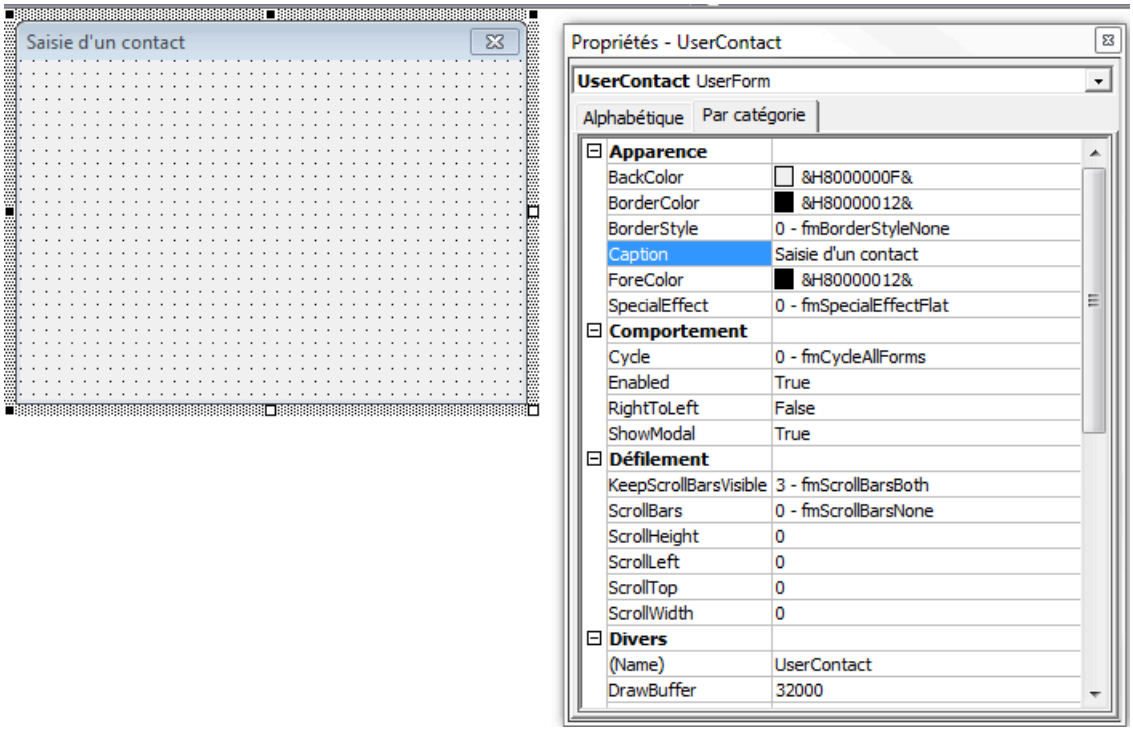
XIV - Programmer en VBA dans un formulaire EXCEL

Nous allons allier la puissance d'un formulaire et celle du VBA, en prenant pour premier exemple la création d'un formulaire pour saisir un nom dans une zone de saisie, associé à un bouton de validation et un bouton d'annulation…

À partir de l'éditeur VBA, choisir dans le menu « Insertion », « UserForm ».

Par défaut le formulaire s'appelle « UserForm1 ». Nous allons le renommer par le menu « Affichage », « Fenêtre Propriétés ». Dans la catégorie « (Name) » remplacez « UserForm1 » par « UserContact », et dans la catégorie « Caption » remplacez « UserForm1 » par « Saisie d'un contact ».

Ce qui donne ce résultat :



Puis choisir dans le menu « Affichage », « Boîte à outils ».

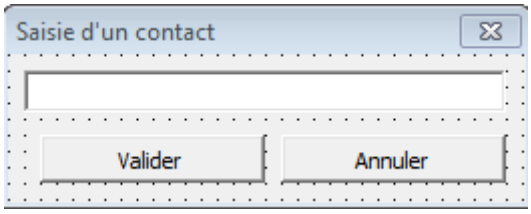
Cliquez sur « Zone de texte », l'icône en forme de « ab| », pour ajouter une zone de saisie. Lorsque la souris passe sur le formulaire, le curseur prend la forme d'un plus. Cliquez sur le formulaire et maintenez le clic gauche enfoncé pour définir la taille de l'outil affiché en pointillés, puis relâchez le clic.

Un clic droit ou « F4 » permet de modifier les propriétés de l'objet créé.

Changez dès maintenant le nom (Name) en « TboNom ».

De la même façon, insérez deux « boutons de commande » dans le formulaire et modifiez leur nom (Name) et libellé (Caption) : Respectivement, « BtnValider » avec en libellé « Valider » et « BtnAnnuler » avec en libellé « Annuler ».

La taille du formulaire peut aussi être modifiée pour obtenir ceci :



Faites un double-clic sur le bouton « BtnValider », l'éditeur de code s'ouvre et a déjà créé l'événement « sur clic » du bouton :

```
Private Sub BtnValider_Click()

End Sub
```

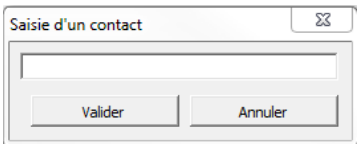
Un clic sur ce bouton va alimenter la variable publique Choix à « Valider » et masquera le formulaire. Un clic sur le bouton « BtnAnnuler » alimentera cette variable à « Annuler ».

Ce qui donne le code suivant :

```
Public Choix As String

Private Sub BtnValider_Click()
Choix = "Valider"
Me.Hide
End Sub

Private Sub BtnAnnuler_Click()
Choix = "Annuler"
Me.Hide
End Sub
```

Le code du formulaire peut être affiché en sélectionnant le formulaire dans l'explorateur de projets [Ctrl]+[R] par un clic droit, puis « Code ».

Pour lancer ce formulaire depuis votre code VBA, utilisez l'instruction **UserContact.Show**.



Par défaut, le formulaire est en mode modal.

La validation d'un des deux boutons, ou de la croix, rendra la main au code appelant. Vous pourrez alors lire le contenu de la variable Choix qui a été définie dans le formulaire par l'instruction **UserContact.Choix** et lire le contenu de la zone de texte « TboNom » par l'instruction **UserContact.TboNom**.

Ce formulaire sera fermé par l'instruction **Unload UserContact**.

Dans un module, copiez le code suivant, qui permet d'afficher en « A1 » le contenu de la saisie, seulement si le bouton « Valider » a été cliqué :

```
'-------------------------------------------------------------------------------
Sub Utiliser_UserContact()
'-------------------------------------------------------------------------------
UserContact.Show
If UserContact.Choix = "Valider" Then Range("A1") = UserContact.TboNom
Unload UserContact
End Sub
```

Il est possible d'attribuer une valeur par défaut à l'objet d'un formulaire. Il faut pour cela charger le formulaire en mémoire par l'instruction **Load**, alimenter l'objet rendu disponible, puis afficher le formulaire par l'instruction **Show**.

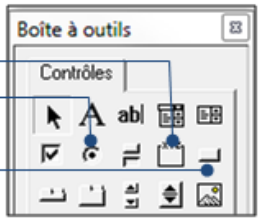
Le code suivant permet d'alimenter la zone de saisie du nom de l'utilisateur actif :

```
'-------------------------------------------------------------------------------
Sub Utiliser_UserContact()
'-------------------------------------------------------------------------------
```

```
Load UserContact
UserContact.TboNom = Application.UserName
UserContact.Show
If UserContact.Choix = "Valider" Then Range("A1") = UserContact.TboNom
Unload UserContact
End Sub
```

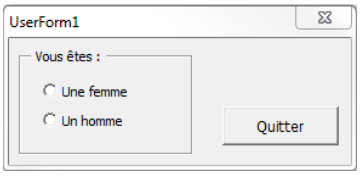
Voici un autre exemple, de création d'un questionnaire pour un sondage…

Insérez un nouveau formulaire. Dans la « Boîte à outils », cliquez sur « Cadre », ici l'icône colonne 4, ligne 2.
Insérez deux « bouton d'option » dans le cadre.
Et un « bouton de commande » à la suite du cadre.



Touche « F5 » pour exécuter le formulaire qui peut ressembler à celui-ci après modification des libellés par défaut :

Les boutons d'option à l'intérieur du cadre permettent de ne sélectionner qu'une seule option, mais à l'ouverture du formulaire, aucune sélection n'est faite.

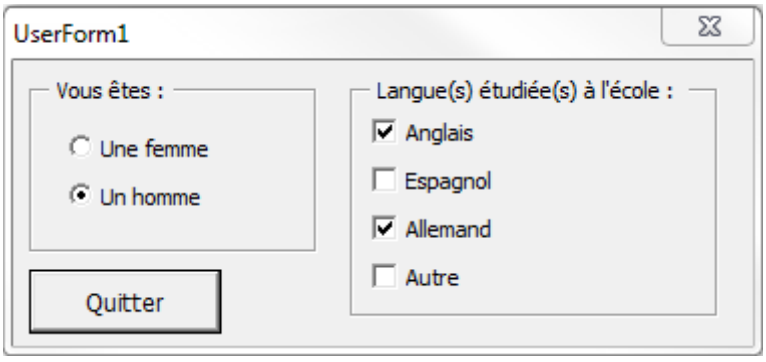


Nous allons demander au formulaire de cocher le bouton d'option nommé « OptionButton1 » à l'ouverture, en utilisant cette fois un événement, « Sur activation » du formulaire (il est cependant conseillé d'utiliser la méthode vue dans l'exemple précédent pour alimenter un objet avant l'ouverture du formulaire).
Faire un clic droit à l'intérieur du formulaire (mais pas sur l'en-tête), puis choisir « code ».

L'éditeur VBA s'ouvre. Piochez dans la liste de gauche « UserForm » et dans la liste de droite « Activate ». Une procédure vierge s'ouvre. Copiez le code suivant :

```
Private Sub UserForm_Activate()
OptionButton1.Value = True
End Sub
```

Ajoutez un nouveau cadre et insérez-y quatre « cases à cocher », qui permettent, elles, de faire des choix multiples :



Nous allons associer un événement « sur clic » au bouton de commande pour enregistrer ces données sur la feuille 6 du classeur, en ligne 2.

Depuis le formulaire faites un double-clic sur le bouton de commande, ou depuis l'éditeur piochez dans la liste de gauche « CommandButton1 » et dans la liste de droite « Click ».

Copiez le code suivant :

```
Private Sub CommandButton1_Click()
Dim y As Integer
y = 2
Sheets("Feuil6").Cells(y, 1) = IIf(OptionButton1.Value = True, "Femme", "Homme")
Sheets("Feuil6").Cells(y, 2) = CheckBox1
Sheets("Feuil6").Cells(y, 3) = CheckBox2
Sheets("Feuil6").Cells(y, 4) = CheckBox3
Sheets("Feuil6").Cells(y, 5) = CheckBox4
UserForm1.Hide
End Sub
```

Pour obtenir ce résultat :

	A	B	C	D	E
1	Civilité	Anglais	Espagnol	Allemand	Autre
2	Homme	VRAI	FAUX	VRAI	FAUX
3					

La fonction **IIf(Expression, ArgumentSiVrai, ArgumentSiFaux)** renvoie l'un ou l'autre des deux arguments selon le résultat de l'évaluation de l'expression. Ici, si le bouton d'option 1 est coché (donc vaut **True**) alors la fonction retourne le texte « Femme », sinon c'est le texte « Homme » qui est retourné.

Cette fonction peut remplacer avantageusement une condition **If … Then**

Ajouter maintenant une zone de liste modifiable qui contiendra la liste des couleurs préférées.

Cette liste est alimentée par les valeurs inscrites dans la colonne « J » de la feuille 6.

J
Liste Couleur
rouge
bleu
vert
jaune

Modifiez le code de l'événement « sur activation » du formulaire pour qu'il charge ces données dans l'objet créé « Combobox1 ».

```
'--------------------------------------------------------------------------------
Private Sub UserForm_Activate()
'--------------------------------------------------------------------------------
Dim y As Integer, i As Integer, TabDonnnée() As Variant

' Coche la case option 1 :
OptionButton1.Value = True

' Charge les données pour la liste ComboBox1 :
y = 2: i = 0
While Sheets("Feuil6").Cells(y, 10) <> "" ' Boucle sur les lignes de la feuille 6.
    ReDim Preserve TabDonnnée(i)          ' Dimensionne le tableau.
    TabDonnnée(i) = Sheets("Feuil6").Cells(y, 10) ' Charge les données en colonne J.
    y = y + 1: i = i + 1                  ' Passe à la ligne suivante.
Wend
Call VBO.TriRapide(TabDonnnée()) ' Trie les données par ordre croissant.
ComboBox1.Clear ' Efface toutes les anciennes données de la liste.
For i = 0 To UBound(TabDonnnée())    ' Boucle sur le tableau et...
    ComboBox1.AddItem TabDonnnée(i) ' alimente la liste déroulante.
Next i
```

```
End Sub
```

Pour récupérer sur la feuille 6 en colonne « F » la valeur choisie dans la liste déroulante :

```
Sheets("Feuil6").Cells(y, 6) = ComboBox1.Value
```

En laissant la propriété MatchRequired de cet objet à False, l'utilisateur peut faire un choix non prévu dans la liste déroulante.

Pour ajouter ce choix à la liste des couleurs existantes, il suffit de l'ajouter à la suite de la liste de la feuille 6 en colonne « J » si elle n'existe pas encore. Elle sera prise en compte immédiatement, car la liste des couleurs est chargée à chaque fois que le formulaire est activé.

Créez cette procédure à la suite de celles existantes :

```
Sub AjouterCouleur()
Dim y As Long
y = 2
While Sheets("Feuil6").Cells(y, 10) <> "" ' Boucle sur les lignes.
    If ComboBox1.Value = Sheets("Feuil6").Cells(y, 10) _
        Then Exit Sub ' Quitte si la couleur existe.
    y = y + 1 ' Passe à la ligne suivante.
Wend
Sheets("Feuil6").Cells(y, 10) = ComboBox1.Value ' Ajoute la couleur.
End Sub
```

Cette procédure sera appelée par l'événement « sur clic » au bouton de commande 1 :

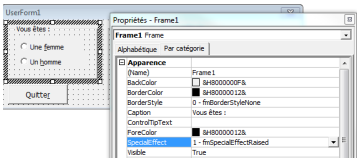
```
Private Sub CommandButton1_Click()
Dim y As Integer
y = 2
Sheets("Feuil6").Cells(y, 1) = IIf(OptionButton1.Value = True, "Femme", "Homme")
Sheets("Feuil6").Cells(y, 2) = CheckBox1
Sheets("Feuil6").Cells(y, 3) = CheckBox2
Sheets("Feuil6").Cells(y, 4) = CheckBox3
Sheets("Feuil6").Cells(y, 5) = CheckBox4
Sheets("Feuil6").Cells(y, 6) = ComboBox1.Value
UserForm1.Hide
Call AjouterCouleur ' Ajoutera la couleur si nécessaire
End Sub
```

Le code d'un formulaire peut contenir des procédures événementielles, ou ordinaires. Faire appel à des procédures publiques, des autres modules de l'application, et aux procédures du formulaire.

Nous n'allons pas étudier un à un les différents contrôles qu'un formulaire peut contenir, car ils fonctionnent tous sur le même principe.

N'hésitez pas à modifier les propriétés des contrôles pour mieux comprendre leur incidence. Touche [F4] pour un accès rapide ou « Affichage », « Fenêtre Propriétés ».

N'oubliez pas l'aide très complète accessible par la touche [F1].



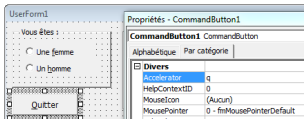
⚠ Attention : dans le code présenté dans ces exemples, les noms des feuilles et des cellules sont écrits en dur, mais dans la pratique il faudra toujours utiliser des variables, au cas où le

nom du classeur change, ou que les colonnes soient déplacées. Nous étudierons plus tard comment faire cela.

Trois astuces.

- Gérer l'ordre de tabulation des contrôles : c'est l'ordre de sélection des contrôles quand l'utilisateur utilise la touche « Tabulation » pour passer d'un contrôle à l'autre. Menu « Affichage », « Ordre de tabulation ». Ça permet de respecter une logique de saisie.
- Affecter une touche d'accès rapide à un contrôle : mettre dans la propriété « Accelerator » la lettre qui permet la validation du contrôle.

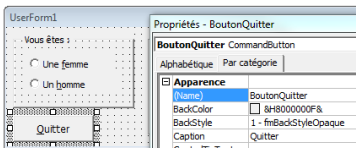
Le libellé du contrôle est souligné automatiquement. Dans notre exemple [Alt]+[q] permet de quitter le formulaire.



[Alt]+[f] permet de sélectionner « Une femme » et [Alt]+[h] « Un homme ».

- Renommer les contrôles pour avoir un code plus lisible : préférez un nom de contrôle parlant, par exemple « BoutonQuitter » de préférence à « CommandButton1 ».

Si vous le faites après coup, n'oubliez pas de renommer les événements liés à ce contrôle, qui malheureusement, ne sont pas renommés automatiquement dans le code du formulaire.



Dans ce cas, remplacez « Private Sub CommandButton1_Click() » par « Private Sub BoutonQuitter_Click() ».

Pour lancer ce formulaire dans votre code VBA, utilisez :

```
Sub Sondage()
UserForm1.Show      ' Lance le formulaire de saisie du sondage.
Unload UserForm1    ' Ferme le formulaire et libère la mémoire.
End Sub
```

Le formulaire est renseigné. Après avoir vérifié que les données saisies sont conformes aux attentes, nous allons demander à l'utilisateur s'il faut ou non sauvegarder ces données.

Lors de nos premiers pas en VBA, nous avons étudié l'instruction **Msgbox** qui permet d'afficher une boîte de dialogue, avec un à trois boutons. Elle est très pratique et simple à utiliser, mais les libellés des boutons ne peuvent pas être personnalisés. C'est pourquoi nous allons créer notre propre boîte de dialogue à trois boutons grâce à un formulaire…

Première étape : créez un formulaire nommé « Boite_3 » qui contient un « Intitulé » et trois « boutons de commande ». Nous allons utiliser la propriété **Tag** (remarque) du formulaire pour y stocker le numéro du bouton de commande choisi, et utiliser cette valeur dans la suite de nos traitements. Sachant que si l'utilisateur clique sur la croix pour fermer le formulaire, cette propriété restera à vide.

Donc, affectez un événement « sur Clic » à chacun de ces boutons de commande, pour alimenter la propriété **Tag** du formulaire « Boite_3 », puis masquer le formulaire :

```
Private Sub CommandButton1_Click()
Me.Tag = "1": Me.Hide
End Sub
Private Sub CommandButton2_Click()
Me.Tag = "2": Me.Hide
End Sub
```

```
Private Sub CommandButton3_Click()
Me.Tag = "3": Me.Hide
End Sub
```

Deuxième étape : pour appeler le formulaire, créez une fonction nommée « Boite_3_Boutons » qui contiendra les arguments nécessaires pour personnaliser notre boîte. À savoir, un titre, un message, le libellé des trois boutons, la hauteur de la boîte, qui dépendra de la hauteur du message.

La fonction retourne la valeur contenue dans la propriété **Tag** du formulaire : soit la valeur zéro, soit une valeur de 1 à 3 suivant le bouton validé.

```
'-------------------------------------------------------------------------------
Function Boite_3_Boutons(Titre, Message, Bouton1, Bouton2, Bouton3, Hauteur) As Byte
'-------------------------------------------------------------------------------
Load Boite_3                                ' Charge le formulaire en mémoire.
Boite_3.Caption = Titre                     ' Le titre de la boite
Boite_3.Label1.Caption = Message            ' Le message à afficher.
Boite_3.CommandButton1.Caption = Bouton1    ' Le libellé du bouton 1.
Boite_3.CommandButton2.Caption = Bouton2    ' Le libellé du bouton 2.
Boite_3.CommandButton3.Caption = Bouton3    ' Le libellé du bouton 3.
Boite_3.Label1.Height = Hauteur             ' La hauteur du message.
Boite_3.Height = Hauteur + 25               ' La hauteur de la boite ajustée au message.
Boite_3.CommandButton1.Top = Hauteur - 25   ' La position du bouton 1 ajustée au message.
Boite_3.CommandButton2.Top = Hauteur - 25   ' La position du bouton 2 ajustée au message.
Boite_3.CommandButton3.Top = Hauteur - 25   ' La position du bouton 3 ajustée au message.
Boite_3.Show                                ' Lance la boîte 3 boutons.
Boite_3_Boutons = Val(Boite_3.Tag) ' Retourne le bouton validé ou 0 si boite fermée.
Unload Boite_3
End Function
```

Ne reste plus qu'à appeler la fonction, puis agir suivant le bouton validé :

```
Titre = "Fin de la saisie..."
Message = "Les données saisies sont compatibles avec le format attendu." _
        & Chr(10) & Chr(13) & "Que voulez-vous faire de ces données ?"
Do
    Select Case Boite_3_Boutons(Titre, Message, "Enregistrer", "Annuler", "À l'aide", 80)
        Case 0, 2: Exit Sub ' Bouton "Annuler" ou fermeture du formulaire = "Quitter"
        Case 3: MsgBox "Cliquez sur le bouton de votre choix." ' Bouton "À l'aide".
        Case 1: Exit Do ' Bouton "Enregistrer" = Sort de la boucle et suite du traitement
    End Select
Loop
```

L'utilisateur a choisi le bouton « Enregistrer », nous allons voir comment enregistrer ces données dans une base ACCESS…

XV - Envoyer des données dans ACCESS - Importer des données d'ACCESS

Nous pouvons communiquer avec ACCESS de la même manière que nous avons communiqué avec OUTLOOK. Commençons par charger les dictionnaires ACCESS, depuis l'éditeur VBA, dans le menu « Outils », « Références », en cochant « Microsoft Access 14.0 Object Library », « Microsoft DAO 3.6 Object Library » et « Microsoft ActiveX Data Objects 6.1 Library ».

Pour un premier exercice, nous allons travailler sur une base ACCESS au format MDB, « Base_Sondage.mdb » qui contient la table « Données » avec sept champs : « Id» au format clé unique automatique, « Civilité » au format texte, « Anglais », « Espagnol », « Allemand », « Autre » au format Oui/Non et « Couleur » au format texte.

Pour enregistrer de nouvelles données, il faut respecter les étapes suivantes : ouvrir la base, ouvrir la table, passer la table en mode ajout, ajouter les données, valider les modifications faites, fermer la table, fermer la base :

```
'-------------------------------------------------------------------------------
Sub EnregistrerAccessMDB()
```

```
'-------------------------------------------------------------------------------
Dim MaBase As Database       ' Objet représentant la base.
Dim MaTable As Recordset     ' Objet représentant la table de la base.
Dim y As Long

Set MaBase = OpenDatabase("C:\_Formation_VBA\Base_Sondage.mdb") ' Ouvre la base.
Set MaTable = MaBase.OpenRecordset("Données", dbOpenTable)      ' Ouvre la table.
y = 2

MaTable.AddNew ' Passe la table en mode Ajout.

MaTable!Civilité = Sheets("Feuil6").Cells(y, 1) ' Alimente le champ Civilité.
MaTable!Anglais = Sheets("Feuil6").Cells(y, 2)  ' Alimente le champ Anglais.
MaTable!Espagnol = Sheets("Feuil6").Cells(y, 3) ' Alimente le champ Espagnol.
MaTable!Allemand = Sheets("Feuil6").Cells(y, 4) ' Alimente le champ Allemand.
MaTable!Autre = Sheets("Feuil6").Cells(y, 5)    ' Alimente le champ Autre.
MaTable!Couleur = Sheets("Feuil6").Cells(y, 6)  ' Alimente le champ Couleur.

MaTable.Update ' Enregistre les modifications faites.

MaTable.Close ' Ferme la table.
MaBase.Close  ' Ferme la base.

End Sub
```

Les champs de la table sont accessibles par un point d'exclamation, à ne pas confondre avec les instructions qui utilisent le point.

La variable nommée MaTable représente en fait un jeu d'enregistrements (Recordset) d'une table. Certains préféreront donc un nom comme MonEnreg ou MonRec plus explicite pour eux.

Inversement, pour lire les données de cette même base on utilise une boucle sur les enregistrements de la table :

```
'-------------------------------------------------------------------------------
Public Sub LireAccessMDB()
'-------------------------------------------------------------------------------
Dim MaBase As Database       ' Objet représentant la base.
Dim MaTable As Recordset     ' Objet représentant la table de la base.
Dim y As Long

Set MaBase = OpenDatabase("C:\_Formation_VBA\Base_Sondage.mdb")            ' Ouvre la base
Set MaTable = MaBase.OpenRecordset("SELECT * FROM Données", dbOpenDynaset) ' Ouvre la table
y = 2

While MaTable.EOF = False ' Boucle tant que la fin de table n'est pas atteinte.

    Sheets("Feuil6").Cells(y, 1) = MaTable!Civilité ' Reporte la donnée dans EXCEL.
    Sheets("Feuil6").Cells(y, 2) = MaTable!Anglais  ' Reporte la donnée dans EXCEL.
    Sheets("Feuil6").Cells(y, 3) = MaTable!Espagnol ' Reporte la donnée dans EXCEL.
    Sheets("Feuil6").Cells(y, 4) = MaTable!Allemand ' Reporte la donnée dans EXCEL.
    Sheets("Feuil6").Cells(y, 5) = MaTable!Autre    ' Reporte la donnée dans EXCEL.
    Sheets("Feuil6").Cells(y, 6) = MaTable!Couleur  ' Reporte la donnée dans EXCEL.

    MaTable.MoveNext ' Enregistrement suivant.
    y = y + 1        ' Ligne suivante.

Wend

MaTable.Close ' Ferme la table
MaBase.Close  ' Ferme la base

End Sub
```

Ceux qui ont des notions de SQL préféreront ouvrir la table avec l'instruction suivante :

```
Set MaTable = MaBase.OpenRecordset("SELECT * FROM Données [WHERE Condition]", dbOpenDynaset)
```

Où [WHERE Condition] représente une condition SQL qui permet de ne sélectionner que les enregistrements de la table qui répondent à la condition.

Par exemple, que les enregistrements où la couleur préférée est gris :

```
" SELECT * FROM Données WHERE Couleur = 'Gris' "
```

En SQL les textes sont entre guillemets simples, pour les distinguer des variables ou des nombres.

Pour le deuxième exercice, nous allons travailler sur une base ACCESS au format ACCDB, le nouveau format ACCESS. Qui permet l'utilisation du nouveau protocole ADO au lieu du protocole DAO.

Pour enregistrer de nouvelles données, il faut respecter les étapes suivantes : ouvrir une connexion à la base en indiquant le fournisseur à utiliser et le côté du curseur, ouvrir la table, passer la table en mode ajout, ajouter les données, valider les modifications faites, fermer la table, fermer la connexion.

Vous trouverez sur le site Developpez.com un cours complet pour « Comprendre les Recordset ADO » de Jean-Marc Rabilloud : « **https://vb.developpez.com/bidou/recordset-ado/** »

Ce qui donne le code suivant :

```
'-------------------------------------------------------------------------------
Sub EnregistrerAccessACCDB()
'-------------------------------------------------------------------------------
Dim Cnn As ADODB.Connection     ' Objet représentant une connexion à la base.
Dim MaTable As ADODB.Recordset  ' Objet représentant la table de la base.
Dim y As Long
Const ACCDB_Fournisseur = "Microsoft.ACE.OLEDB.12.0"          ' Fournisseur de données.
Const Fichier_Source = "C:\_Formation_VBA\Base_Sondage.accdb" ' Fichier accdb ou mdb.

Set Cnn = New ADODB.Connection   ' Initialisation d'un objet connexion.
Cnn.CursorLocation = adUseServer ' Déclaration du curseur côté serveur.

' Initialisation de la connexion à la base avec déclaration du fournisseur :
Cnn.Open "Provider= " & ACCDB_Fournisseur & ";" _
            & "Data Source=" & ACCDB_Source _
            & ";", "Admin", "", adAsyncConnect
While (Cnn.State = adStateConnecting): DoEvents: Wend ' Attente de la connexion.

Set MaTable = New ADODB.Recordset ' Initialisation d'un objet table.
' Ouverture de la table en lecture/écriture :
MaTable.Open "SELECT * FROM Données", Cnn, adOpenKeyset, adLockPessimistic, adCmdText
y = 2

MaTable.AddNew ' Passe la table en mode Ajout.

MaTable!Civilité = Sheets("Feuil6").Cells(y, 1) ' Alimente le champ Civilité.
MaTable!Anglais = Sheets("Feuil6").Cells(y, 2)  ' Alimente le champ Anglais.
MaTable!Espagnol = Sheets("Feuil6").Cells(y, 3) ' Alimente le champ Espagnol.
MaTable!Allemand = Sheets("Feuil6").Cells(y, 4) ' Alimente le champ Allemand.
MaTable!Autre = Sheets("Feuil6").Cells(y, 5)    ' Alimente le champ Autre.
MaTable!Couleur = Sheets("Feuil6").Cells(y, 6)  ' Alimente le champ Couleur.

MaTable.Update ' Enregistre les modifications faites.

MaTable.Close           ' Ferme la table.
Set MaTable = Nothing   ' Libère l'objet de la mémoire.
Cnn.Close               ' Ferme la connexion.
Set Cnn = Nothing       ' Libère l'objet de la mémoire.

End Sub
```

Le code est plus lourd qu'avec une base MDB, et ici c'est simplifié, mais bonne nouvelle, il est compatible avec les deux formats ACCDB et MDB.

Il est donc plus judicieux pour vous de retenir cette méthode moderne, quel que soit le format de la base ACCESS à traiter. Mais comme l'on voit encore, et peut-être pour de nombreuses années, circuler du code de l'ancienne méthode, nous ne pouvions pas faire l'impasse.

Notez ici l'exemple de l'utilisation d'une constante, où ACCDB_Fournisseur représente le fournisseur de données. Cette constante peut être déclarée publique en en-tête d'un module pour être utilisée par toutes les procédures de l'application.

Et pour lire les données :

```
'-------------------------------------------------------------------------------
Sub LireAccessACCDB()
'-------------------------------------------------------------------------------
Dim Cnn As ADODB.Connection     ' Objet représentant une connexion à la base.
Dim MaTable As ADODB.Recordset  ' Objet représentant la table de la base.
Dim y As Long, x As Long
Const ACCDB_Fournisseur = "Microsoft.ACE.OLEDB.12.0"           ' Fournisseur de données.
Const Fichier_Source = "C:\_Formation_VBA\Base_Sondage.accdb"  ' Fichier accdb ou mdb.

Set Cnn = New ADODB.Connection   ' Initialisation d'un objet connexion.
Cnn.CursorLocation = adUseServer ' Déclaration du curseur cote serveur.

' Initialisation de la connexion à la base avec déclaration du fournisseur :
Cnn.Open "Provider= " & ACCDB_Fournisseur & ";" _
            & "Data Source=" & Fichier_Source _
            & ";", "Admin", "", adAsyncConnect
While (Cnn.State = adStateConnecting): DoEvents: Wend ' Attente de la connexion.

Set MaTable = New ADODB.Recordset ' Initialisation d'un objet table.
' Ouverture de la table en lecture seule :
MaTable.Open "SELECT * FROM Données", Cnn, adOpenKeyset, adLockReadOnly, adCmdText

y = 2
While MaTable.EOF = False ' Boucle tant que la fin de table n'est pas atteinte.
    ' Boucle sur les champs de la table (sauf la clé Id en 0) :
    For x = 1 To MaTable.Fields.Count - 1
        Sheets("Feuil6").Cells(y, x) = MaTable.Fields(x).Value
    Next x

    MaTable.MoveNext ' Enregistrement suivant.
    y = y + 1        ' Ligne suivante.
Wend

MaTable.Close           ' Ferme la table.
Set MaTable = Nothing   ' Libère l'objet de la mémoire.
Cnn.Close               ' Ferme la connexion.
Set Cnn = Nothing       ' Libère l'objet de la mémoire.
End Sub
```

Pour changer, j'ai utilisé une boucle sur les champs de la table pour les afficher dans EXCEL, au lieu de les nommer respectivement. Le champ 0 qui contient dans ma table la clé de l'enregistrement ne m'intéressant pas, je l'ai délibérément ignoré, la boucle commençant au champ 1.

L'argument **SELECT * FROM NomTable** de l'instruction **Open** permet de sélectionner tous les enregistrements de la table NomTable. Cet argument peut supporter une requête SQL comme vu précédemment. Par exemple : "SELECT * FROM Données WHERE Couleur = 'Gris' ".

Depuis EXCEL, il sera plus fréquent de lire des enregistrements d'une base que d'en ajouter. Nous allons donc créer une nouvelle procédure dans notre boîte à outils, pour ouvrir une table d'une base ACCESS, importer les enregistrements dans une feuille choisie, la créer au besoin, et afficher en première ligne le nom des champs de la table. La fonction retourne le nombre d'enregistrements importés, ou -1 en cas d'erreur.

Sheets(StrFeuillle).Range("A2").CopyFromRecordset MaTable est utilisé pour afficher d'un coup tous les enregistrements de la table MaTable vers la cellule « A2 » de la feuille StrFeuille.

```
'-------------------------------------------------------------------------------
Function LireTableAccess(StrBase As String, StrTable As String, SQLWhere As String, _
                         StrFeuillle As String) As Long
'-------------------------------------------------------------------------------
Dim Cnn As ADODB.Connection     ' Objet représentant une connexion à la base.
Dim MaTable As ADODB.Recordset  ' Objet représentant la table de la base.
Dim i As Long, FeuilleActive As String  ' Variables de la procédure.

On Error GoTo Err_Feuille               ' Gestion des erreurs si feuille à créer.
Sheets(StrFeuillle).Cells.ClearContents ' Efface les données de toute la feuille.

On Error GoTo Err_Lecture        ' Gestion des erreurs pendant la lecture.
Set Cnn = New ADODB.Connection   ' Initialisation d'un objet connexion.
Cnn.CursorLocation = adUseServer ' Déclaration du curseur côté serveur.
' Initialisation de la connexion à la base avec déclaration du fournisseur :
Cnn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & "Data Source=" & StrBase _
            & ";", "Admin", "", adAsyncConnect
While (Cnn.State = adStateConnecting): DoEvents: Wend ' Attente de la connexion.

Set MaTable = New ADODB.Recordset ' Initialisation d'un objet table.
' Ouverture de la table, avec ou sans requête SQLWhere, en lecture seule :
MaTable.Open "SELECT * FROM [" & StrTable & "]" & IIf(SQLWhere > "", " WHERE " & SQLWhere, "") _
             , Cnn, adOpenKeyset, adLockReadOnly, adCmdText

For i = 0 To MaTable.Fields.Count - 1 ' Boucle sur le nom des champs de la table.
    Sheets(StrFeuillle).Cells(1, i + 1).Value = MaTable.Fields(i).Name ' En-tête.
Next

Sheets(StrFeuillle).Range("A2").CopyFromRecordset MaTable ' Affiche les données depuis A2.
LireTableAccess = MaTable.RecordCount ' Retourne le nombre de données lues.

Fermeture:
On Error Resume Next
MaTable.Close           ' Ferme la table.
Set MaTable = Nothing   ' Libère l'objet de la mémoire.
Cnn.Close               ' Ferme la connexion.
Set Cnn = Nothing       ' Libère l'objet de la mémoire.
Err.Clear               ' Efface les éventuelles erreurs.
Exit Function           ' Sort de la fonction

Err_Lecture:            ' Étiquette branchée en cas d'erreur de lecture
MsgBox Err.Description, vbCritical + vbOKOnly, "Erreur" ' Message d'erreur.
LireTableAccess = -1    ' Retourne -1.
Resume Fermeture        ' Branchement pour quitter proprement.

Err_Feuille:                              ' Étiquette branchée si la feuille n'existe pas.
FeuilleActive = ActiveSheet.Name          ' Mémorise la feuille active
Sheets.Add After:=Sheets(Sheets.Count)    ' Création d'une nouvelle feuille.
Sheets(ActiveSheet.Name).Name = StrFeuillle ' Renomme la feuille créée.
Sheets(FeuilleActive).Select              ' Active l'ancienne feuille.
Resume                                    ' Suite du traitement.

End Function
```

Cette procédure peut être appelée ainsi :

```
' Sélection de la base Base_Sondage.accdb :
MaBase = VBO.BoiteDialogue(msoFileDialogFilePicker, "Choisir Base_Sondage", _
                                  "", "", "Base_Sondage.accdb", "ACCESS")

' Importe les enregistrements de la table Données dans Feuil7 et indique leur nombre :
lk = VBO.LireTableAccess(MaBase, "Données", "", "Feuil7")
If lk > -1 Then MsgBox lk & " enregistrements importés.", vbOKOnly, "Fin du traitement"
```

XVI - Utiliser les macros

Pour mettre en jaune la couleur du texte d'une cellule, sur une feuille d'un classeur de l'application EXCEL, il faut respecter la hiérarchie des objets et des propriétés :

Workbooks - Sheets - Range - Font - Color = MaCouleur

Certains objets, propriétés et valeurs, sont bien connus, car nous les utilisons régulièrement, mais nous ignorons, au contraire, la majorité des autres.
Le plus rapide, dans ce cas, est d'enregistrer une macro, analyser le code généré, et récupérer la partie qui nous intéresse.

Par exemple, cliquez dans le menu sur « Développeur », puis « Enregistrer une macro ».

Mettez en jaune le texte de la cellule active. Puis dans le menu « Développeur », cliquez sur « Arrêter l'enregistrement ».

Dans l'éditeur VBA, consultez le code généré :

```
Sub Macro1()
'
' Macro1 Macro
'

'
    With Selection.Font
        .Color = 65535
        .TintAndShade = 0
    End With
End Sub
```

EXCEL a renseigné la propriété **Color** avec la valeur 65535. Ça suffit pour nous donner les informations désirées pour mettre en jaune le texte de la cellule « A1 » de la feuille « Test » de notre application :

```
ThisWorkbook.Sheets("Test").Range("A1").Font.Color = 65535
```

N'hésitez pas à utiliser les macros, pour découvrir les objets utilisés, leurs propriétés et leurs méthodes, et pour vous inspirer du code généré.

Les macros sont des procédures comme les autres, et peuvent être appelées par vos procédures.
Inversement, vous pouvez modifier le code d'une macro pour qu'il fasse appel à vos procédures.

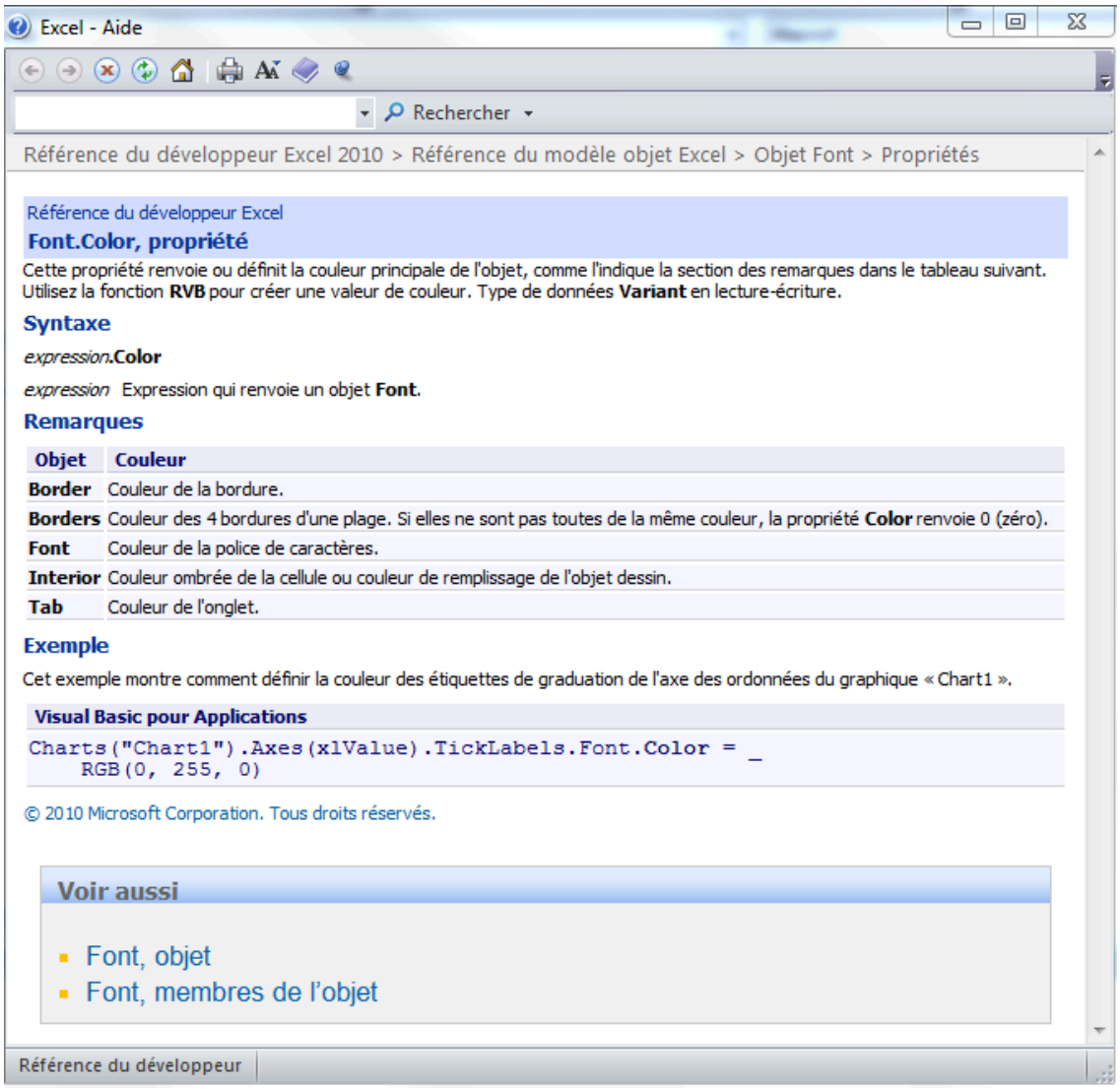
Usez et abusez des macros pour les mises en forme des cellules : colorier, encadrer, fusionner, nécessitent de nombreuses lignes de code qui demandent du temps à écrire, alors qu'une macro produit le résultat attendu en quelques secondes.

Le VBA peut aussi être utilisé pour simplement automatiser des processus développés à l'aide de macros.

Dans le code généré par la macro, cliquez sur **Color** et lancez l'aide avec la touche [F1].
Nous allons apprendre à exploiter l'aide précieuse d'EXCEL.

XVII - Utiliser l'aide

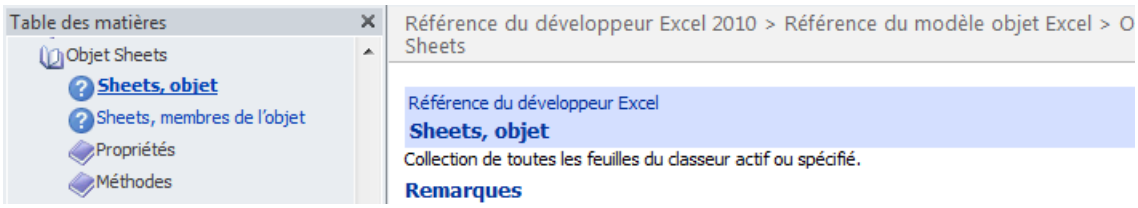
L'aide d'EXCEL est notre principal allié. Pour bien l'exploiter, une explication de texte s'impose.



- Historique des recherches - ou lance une recherche sur le mot saisi.
- Titre : **Font.Color** est bien la propriété demandée.
- Définition : elle nous apprend que cette propriété peut être lue, ou modifiée. Que l'on peut utiliser la fonction RGB pour alimenter la valeur de la propriété, mais passe sous silence que l'on peut aussi utiliser directement un nombre.
- Syntaxe : nous indique qu'il faut utiliser cette propriété avec un objet.
- Remarques : donne la liste des objets qui utilisent cette propriété. L'on retrouve notre objet **font** et l'on apprend que pour modifier la couleur de fond, il faudra utiliser l'objet **Interior** au lieu de **Font**.
- Exemple : permet de se faire une idée de la syntaxe, même si l'exemple ne correspond pas à notre cas.
- Voir aussi : permet d'avoir une aide sur l'objet **Font** qui utilise la propriété, ou d'avoir la liste de tous les autres membres de l'objet **Font** (méthodes, propriétés), où l'on retrouvera notre propriété **Color**.

Vous pouvez afficher l'aide sous forme de table des matières, ce qui est très pratique.

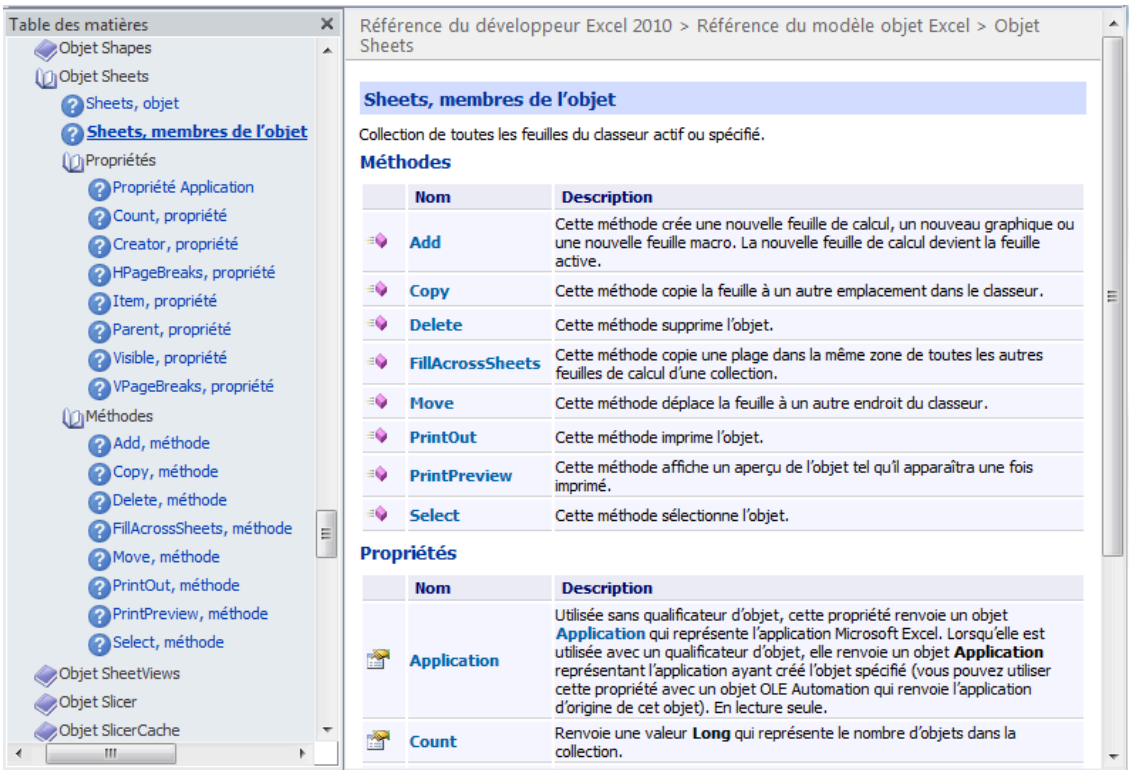
Prenons l'objet **Sheets** que nous connaissons bien maintenant :



Un objet est toujours expliqué en deux parties : première partie, une définition de l'objet. Deuxième partie, une définition des membres de l'objet (les méthodes, les propriétés, et les événements de l'objet).

La partie active est mise en gras.

Ouvre la liste des membres de l'objet : méthodes, propriétés et événements.



- Une méthode permet de faire une action. Par exemple, la méthode **Delete** supprime l'objet.

  `Sheets("Test").Delete` supprime la feuille « Test » du classeur actif.

- Une propriété renvoie ou définit une valeur. Par exemple, la propriété **Count** renvoie le nombre d'objets.

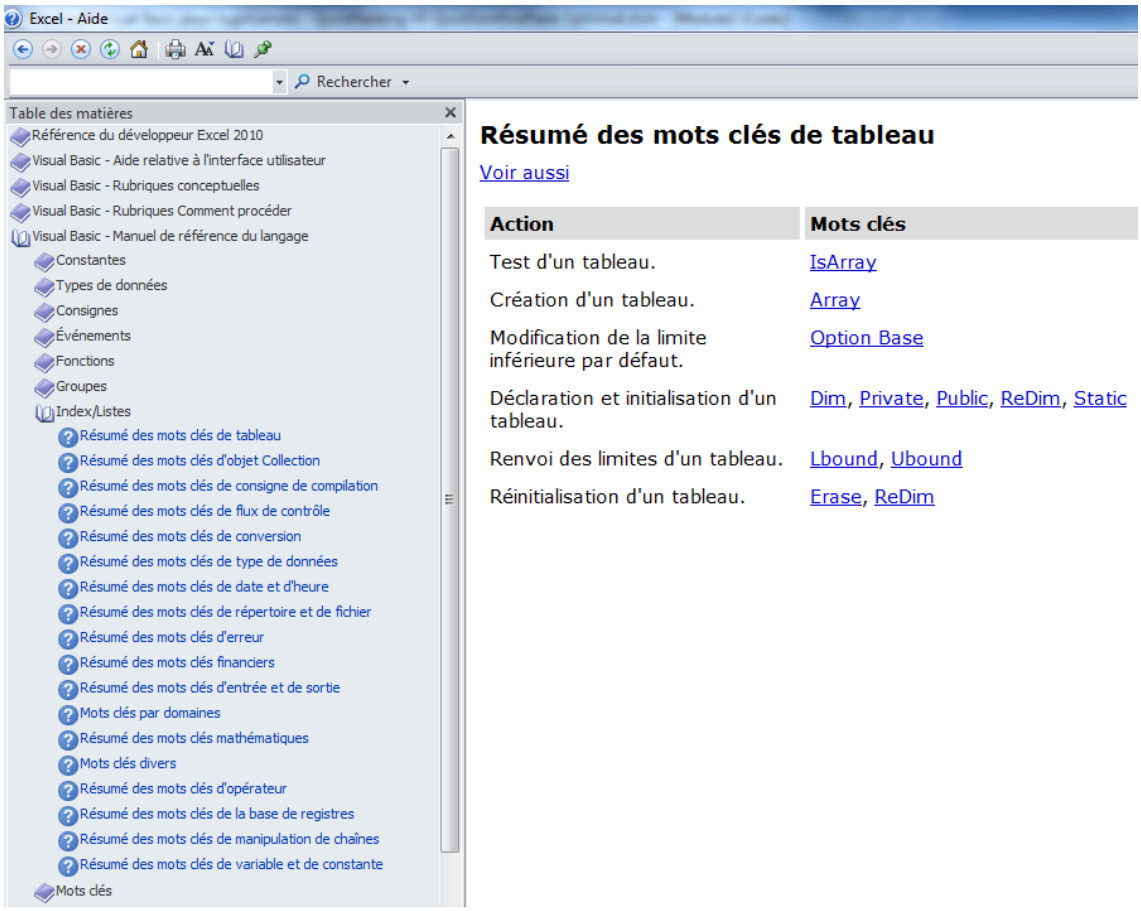
  `Nb = Sheets.Count` renvoie dans la variable `Nb` le nombre de feuilles dans le classeur actif.

- Les événements concernent principalement les objets **Workbooks** et **Worksheet** que nous avons déjà étudiés. Nous allons y revenir pour voir comment personnaliser un classeur à son ouverture.

Mais avant, un dernier conseil sur l'aide d'EXCEL.

L'aide EXCEL contient une rubrique particulièrement intéressante et bien faite : « Visual Basic - Manuel de référence du langage ». Vous retrouverez tous les mots clés du VBA, classés par ordre alphabétique, mais aussi un classement par domaine d'utilisation de ces mots clés.

Par exemple, le résumé des mots clés utilisés lorsque vous manipulez des tableaux de données :



Un clic sur le lien hypertexte du mot clé renvoie sur sa définition.
Prenez le temps de naviguer dans ces différentes listes, de découvrir les possibilités que le VBA peut vous offrir. Le but n'est pas de tout connaître par cœur, mais de savoir que la fonctionnalité existe. En cas de besoin, vous retrouverez facilement avec l'aide le mot clé concerné.

Si vous ne retrouvez pas un mot clé dans la rubrique « Fonctions », pensez à consulter les autres rubriques : « Mots clés », « Instructions ».

Refermons cette parenthèse pour étudier dans les pages qui suivent comment personnaliser un classeur à son ouverture.

XVIII - Personnaliser un classeur

Pour personnaliser un classeur, nous allons intervenir sur les propriétés de l'objet **Application** et de l'objet **ActiveWindow** (fenêtre active).

Certaines propriétés concernent la configuration du classeur :

- la taille de la fenêtre : **Application.WindowState** = énumération **XlWindowState** ;
- l'affichage de la barre des formules : **Application.DisplayFormulaBar** = Vrai/Faux ;
- l'affichage des onglets : **ActiveWindow.DisplayWorkbookTabs** = Vrai/Faux ;
- le mode plein écran : **Application.DisplayFullScreen** = Vrai/Faux.

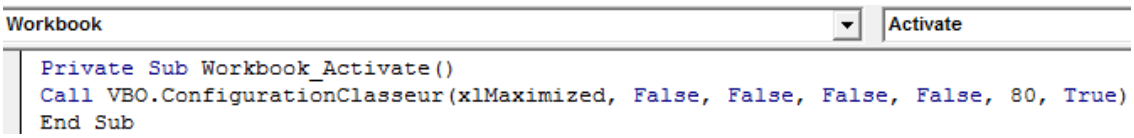
D'autres propriétés concernent la configuration des feuilles du classeur. Il faudra faire une boucle sur les feuilles du classeur pour les activer afin de modifier les propriétés de chacune des feuilles :

- l'affichage de l'en-tête des lignes et colonnes : **ActiveWindow.DisplayHeadings** = Vrai/Faux ;
- l'affichage du quadrillage : **ActiveWindow.DisplayGridlines** = Vrai/Faux ;
- le niveau du zoom : **ActiveWindow.Zoom** = valeur du zoom (100 signifie 100 % = taille normale).

Une procédure personnelle, avec en arguments la valeur à appliquer aux différentes propriétés présentées ici, permet de simplifier la configuration du classeur. Elle sera codée dans le module VBO :

```
'-------------------------------------------------------------------------------
Sub ConfigurationClasseur(Optional EtatFenetre As XlWindowState = xlMaximized, _
    Optional BarreFormule As Boolean = True, Optional Onglet As Boolean = True, _
    Optional Entete As Boolean = True, Optional Quadrillage As Boolean = True, _
    Optional Zoom As Integer = 100, Optional PleinEcran As Boolean = False, _
    Optional ReferenceStyle As XlReferenceStyle = xlA1)
'-------------------------------------------------------------------------------
Dim ActiveS As Worksheet, S As Worksheet
On Error Resume Next

Application.WindowState = EtatFenetre            ' État du classeur.
Application.DisplayFormulaBar = BarreFormule     ' État de la barre des formules.
Application.ReferenceStyle = ReferenceStyle      ' Affiche les colonnes en lettres ou chiffres
ActiveWindow.DisplayWorkbookTabs = Onglet        ' État des onglets des feuilles.
Application.DisplayFullScreen = PleinEcran       ' État du plein écran.
Application.ScreenUpdating = False               ' Bloque la Mise à jour de l'écran.
Set ActiveS = ActiveSheet                        ' Mémorise la feuille active
For Each S In ThisWorkbook.Sheets                ' Boucle sur les feuilles...
    S.Activate                                   ' Active la feuille.
    ActiveWindow.WindowState = EtatFenetre       ' État de la fenêtre.
    ActiveWindow.DisplayHeadings = Entete        ' État des en-têtes.
    ActiveWindow.DisplayGridlines = Quadrillage  ' État du quadrillage.
    ActiveWindow.Zoom = Zoom                     ' Valeur du zoom.
Next S
ActiveS.Activate                    ' Revient sur l'ancienne feuille active.
Application.ScreenUpdating = True   ' Libère la mise à jour de l'écran.
End Sub
```

Dans l'exemple qui suit, l'événement « Sur Activation » du classeur est utilisé, afin de configurer la présentation du classeur dès son activation :



De même, vous pouvez forcer le mode plein écran, que l'utilisateur quitte avec la touche [Echap], en lançant la configuration personnelle depuis l'événement « Sur fenêtre redimensionnée » du classeur.

Pour restaurer la configuration par défaut, un appel suffit : **VBO.ConfigurationClasseur**
Pour personnaliser un peu plus le classeur, nous allons ajouter un menu « Compléments » au ruban d'EXCEL. Ce menu regroupera une ou plusieurs barres de commandes personnelles. Une barre de commandes contiendra des contrôles de type bouton, menu contextuel, liste déroulante, liste modifiable, zone de saisie, pour lancer nos procédures.

Nous allons utiliser l'objet **CommandBars**, car la nouvelle méthode « *Office Fluent* » disponible depuis Office 2007, certes plus puissante, car elle permet de gérer l'ensemble du ruban, est trop complexe à utiliser. Pour plus d'informations, recherchez le mot clé « Ruban » dans l'aide d'EXCEL.

Pour créer une nouvelle barre de commandes personnelle :

```
Set MaBarre = Application.CommandBars.Add("MaBarrePerso", msoBarTop, False, True)
MaBarre.Visible = True
```

Le menu « Compléments » sera ajouté automatiquement au ruban. Inversement, il sera automatiquement supprimé du ruban s'il n'existe plus aucune barre de commandes.

Nous pouvons maintenant ajouter des contrôles à la barre de commandes créée. Ils seront affichés à droite l'un de l'autre. Si l'effet visuel ne vous convient pas, vous pouvez créer une nouvelle barre de commandes pour chaque contrôle. Ainsi les contrôles seront affichés l'un sous l'autre.

Pour ajouter à cette barre de commandes un contrôle de type MsoControlType (énumération) :

```
Set MonMenu = MaBarre.Controls.Add(Type:=MsoControlType)
```

- Bouton : **msoControlButton** ;
- Menu contextuel : **msoControlPopup** ;
- Zone de saisie : **msoControlEdit** ;
- Liste déroulante : **msoControlDropdown** ;
- Liste modifiable : **msoControlComboBox**.

Il faut ensuite alimenter les propriétés du contrôle créé (il n'y a pas d'ordre à respecter) :

- Icône (pour les contrôles de type bouton) : **MonMenu.FaceId =** Numéro de l'icône ;
- Style (pour les contrôles de type bouton) : **MonMenu.Style = msoButtonIcon** (icône seule) ou **msoButtonIconAndCaption** (icône et libellé) ;
- Libellé : **MonMenu.Caption =** Texte ;
- Info-bulle : **MonMenu.TooltipText =** Texte ;
- Nom de la procédure à exécuter : **MonMenu.OnAction =** MaProcédure.

Si ce contrôle est de type zone de liste ou liste modifiable, il vous faut ajouter les éléments du contrôle :

- **MonMenu.AddItem** Libellé_Elément_1 ;
- **MonMenu.AddItem** Libellé_Elément_2 ;
- **…**

Ajoutez ainsi tous les éléments désirés.

Si le contrôle créé est de type menu contextuel, il faut ajouter les options du contrôle :

- `Set MonControle = MonMenu.Controls.Add(msoControlButton)`

  Libellé : **MonControle.Caption =** Texte_Option_1.

  Nom de la procédure à exécuter : **MonControle.OnAction =** *Procédure_Option_1*.

- `Set MonControle = MonMenu.Controls.Add(msoControlButton)`

  Libellé : **MonControle.Caption =** Texte_Option_2.

  Nom de la procédure à exécuter : **MonControle.OnAction =** *Procédure_Option_2*.

…

Ajoutez ainsi toutes les options désirées au menu contextuel.

Pour supprimer une barre de commandes :

- Vous pouvez utiliser la méthode **Delete** de l'objet **CommandBars :**
  Application.CommandBars("MaBarrePerso").Delete

- ou utiliser la fonction VBO.MenuSupprimer(NomMenu As String) avec en argument le nom du menu à supprimer. Si l'argument est une chaîne vide, tous les menus personnels sont supprimés.

La fonction pour supprimer une barre de commandes :

```
'-------------------------------------------------------------------------------
Sub MenuSupprimer(NomMenu As String)
'-------------------------------------------------------------------------------
On Error Resume Next
Dim Cb As CommandBar
For Each Cb In Application.CommandBars
    If Cb.BuiltIn = False And (Cb.Name = NomMenu Or NomMenu = "") Then
        Cb.Delete
        If NomMenu <> "" Then Exit For
    End If
Next Cb
End Sub
```

Le menu « Compléments » est actualisé automatiquement.
Vous pouvez donc supprimer ou ajouter des menus personnels, qui s'adaptent à vos besoins, en cours d'application.

Une procédure personnelle nommée « MenuAjouter » codée dans le module « VBO » permet de simplifier la création d'un menu personnel.

Cinq arguments sont nécessaires :

- TypeMenu : est le type du contrôle, c'est l'énumération MsoControlType ;
- NomMenu : est utilisé comme nom de la barre de commandes et nom du contrôle. Pour les boutons et les menus contextuels, il représente aussi le libellé affiché ;
- ElémentMenu : pour les contrôles du type zone de liste ou liste modifiable, c'est l'élément du contrôle. Pour les contrôles du bouton ou zone de saisie, qui n'ont pas d'élément, c'est le libellé de l'info-bulle ;
- ProcédureLiée : est le nom de la procédure à exécuter quand le contrôle est activé par un clic sur un bouton, la sélection d'un élément, ou une saisie, suivant le type du contrôle ;
- IconeMenu : est le numéro de l'icône à afficher, valable pour les boutons uniquement.

La fonction pour créer un menu personnel :

```
'-------------------------------------------------------------------------------
Sub MenuAjouter(TypeMenu As MsoControlType, NomMenu As String, _
    ElémentMenu As String, ProcédureLiée As String, Optional IconeMenu As Long = 0)
'-------------------------------------------------------------------------------
Dim MaBarre As CommandBar
Dim MonMenu, MonControle
Dim i As Integer

' Création d'une barre de commandes personnelle si elle n'existe pas déjà :
On Error Resume Next
Set MaBarre = Application.CommandBars.Add(NomMenu, msoBarTop, False, True)
Set MaBarre = Application.CommandBars(NomMenu) ' Sélectionne la barre.
MaBarre.Visible = True                         ' Affiche la barre.

' Ajouter le Menu demandé (si besoin) :
For i = 1 To MaBarre.Controls.Count             ' Boucle sur les menus existants.
    Set MonMenu = MaBarre.Controls(i)           ' Sélectionne le menu.
```

```
    If MonMenu.Caption = NomMenu Then GoTo Suite ' Si c'est celui demandé alors sort.
Next i
Set MonMenu = MaBarre.Controls.Add(Type:=TypeMenu) ' Sinon création.
MonMenu.Caption = NomMenu                          ' Le libellé est le nom du menu.
MonMenu.TooltipText = " "                          ' Efface l'info-bulle.
Suite:

' Si le menu est un menu contextuel, ajoute le contrôle passé dans ElémentMenu :
If TypeMenu = msoControlPopup Then
    Set MonControle = MonMenu.Controls.Add(msoControlButton) ' Création.
    MonControle.Caption = ElémentMenu                        ' Texte = Libellé.
    MonControle.OnAction = ProcédureLiée                     ' Procédure liée.
    Exit Sub                                                 ' Fin du traitement.
End If

' Si le menu est une zone de liste ou une liste modifiable :
If TypeMenu = msoControlDropdown Or TypeMenu = msoControlComboBox Then
    MonMenu.AddItem ElémentMenu    ' Ajoute le contrôle passé dans ElémentMenu.
    ElémentMenu = "..."            ' Efface l'info-bulle.
End If

' Autres cas, renseigne les propriétés du menu :
MonMenu.FaceId = IconeMenu                                    ' Icône du menu.
MonMenu.Style = IIf(ElémentMenu > "", msoButtonIconAndCaption, msoButtonIcon)
MonMenu.OnAction = ProcédureLiée                              ' Prodédure liée.
MonMenu.TooltipText = IIf(ElémentMenu > "", ElémentMenu, " ") ' Info-bulle.

End Sub
```

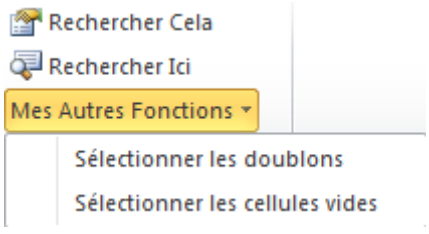
Le menu personnel est provisoire, il est donc automatiquement supprimé du ruban lorsque vous quittez EXCEL. Par contre, le menu personnel reste accessible aux autres classeurs ouverts. Dans vos développements, pensez aux conséquences si votre menu est activé depuis un autre classeur. Ou si un autre classeur possède lui aussi un menu personnel.

Vous pouvez supprimer votre menu personnel en utilisant l'événement « Sur Désactivation » du classeur, s'il ne doit pas être utilisé par les autres classeurs :

```
Private Sub WorkBook_Deactivate()
Call VBO.MenuSupprimer("")
End Sub
```

Exemple pour créer un menu personnel avec deux boutons et un menu contextuel contenant deux éléments :

```
Call VBO.MenuAjouter(msoControlButton, "Rechercher Cela", "1re étape", "RechCela", 222)
Call VBO.MenuAjouter(msoControlButton, "Rechercher Ici", "2e étape", "RechIci", 1561)
Call VBO.MenuAjouter(msoControlPopup, "Mes Autres Fonctions", "Sélectionner les
 doublons", "SelDoublons")
Call VBO.MenuAjouter(msoControlPopup, "Mes Autres Fonctions", "Sélectionner les cellules
 vides", "SelVides")
```



Exemple pour créer une zone de saisie :

```
Call VBO.MenuAjouter(msoControlEdit, "MaSaisie", "Saisir un texte", "ActionTexte")
```

La fonction « ActionTexte » est exécutée si le texte de la zone de saisie est modifié.
Ce texte peut être lu par : `Application.CommandBars("MaSaisie").Controls(1).Text`

Vous pouvez modifier le texte de cette zone : Application.CommandBars("MaSaisie").Controls(1).Text = ""

Exemple pour créer une zone de liste avec trois éléments :

```
Call VBO.MenuAjouter(msoControlDropdown, "MaListe", "Ligne 1", "ActionListe")
Call VBO.MenuAjouter(msoControlDropdown, "MaListe", "Ligne 2", "ActionListe")
Call VBO.MenuAjouter(msoControlDropdown, "MaListe", "Ligne 3", "ActionListe")
```

La fonction « ActionListe » est exécutée si un élément est sélectionné.
Cet élément peut être lu par : Application.CommandBars("MaListe").Controls(1).Text
Vous ne pouvez pas modifier le texte de cette zone.

Exemple pour créer une liste modifiable avec trois éléments :

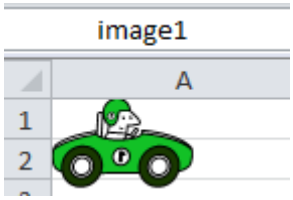
```
Call VBO.MenuAjouter(msoControlComboBox, "ListeMod", "Ligne 1er élément", "ActionLM")
Call VBO.MenuAjouter(msoControlComboBox, "ListeMod", "Ligne 2e élément", "ActionLM")
Call VBO.MenuAjouter(msoControlComboBox, "ListeMod", "Ligne 3e élément", "ActionLM")
```

La fonction « ActionLM » est exécutée si un élément est sélectionné ou si un texte est saisi.
Ce texte peut être lu par : Application.CommandBars("ListMod").Controls(1).Text
Vous pouvez modifier le texte de cette liste : Application.CommandBars("ListMod").Controls(1).Text = ""

Nous avons vu comment personnaliser la présentation du classeur. Une application peut aussi être facilement personnalisée. Ici je vous propose une procédure qui met un peu d'animation dans EXCEL.

Depuis EXCEL, insérez cette image clipart sur une nouvelle feuille :

Affectez-lui une rotation horizontale pour que le conducteur soit dans l'autre sens et placez l'image en haut à gauche de la feuille. Vous obtenez cela :



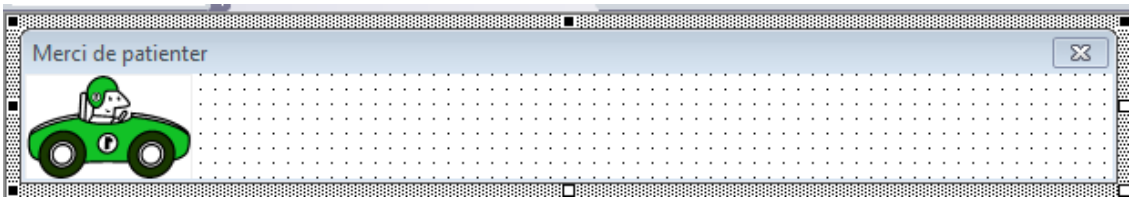
Insérez le code suivant dans un nouveau module pour faire une animation :

```
Sub DéplaceImage(Img As Integer)
Dim i As Integer, T As Double

For i = 1 To 26 ' Boucle pour faire une rotation de la voiture.
   ActiveSheet.Shapes(Img).IncrementRotation (i) ' Rotation.
   T = Timer: While T + 0.05 > Timer: Wend ' Pour faire une pause.
   DoEvents                                ' Actualise l'écran.
Next i
ActiveSheet.Shapes(Img).Rotation = 0  ' Remet à 0.

For i = 1 To 300 ' Boucle pour déplacer horizontalement la voiture.
   ActiveSheet.Shapes(Img).IncrementLeft 1  ' Déplace d'un pixel à droite.
   T = Timer: While T + 0.01 > Timer: Wend  ' Pour faire une pause.
   DoEvents                                 ' Actualise l'écran.
```

```
Next i

End Sub
```

Lancez l'animation par Call DéplaceImage(i) où i est le numéro de l'image (vaut 1 pour la première image créée). Vous pouvez ainsi animer plusieurs images avec cette procédure.

Sur le même principe, vous pouvez facilement faire une barre de progression originale.
Ajoutez un formulaire, nommé « BarreProgression » qui contient l'image de votre choix, comme ici :



L'affichage du formulaire se fait avec Call BarreProgression.Show(False) où False rend le formulaire non modal, votre code garde ainsi la main. La progression se fait avec les instructions suivantes :

```
BarreProgression.Image1.Left = (BarreProgression.Width - BarreProgression.Image1.Width) * x
BarreProgression.Repaint
```

Où x est un pourcentage entre 0 et 1, et l'image est nommée « Image1 ».

À la fin du traitement, fermez la barre de progression avec l'instruction Unload BarreProgression.

Maintenant que nous maîtrisons notre classeur, nous allons voir comment manipuler les données dans différents classeurs sans s'emmêler les pinceaux.

XIX - Manipuler les données dans différents classeurs

Nous pouvons facilement faire des manipulations dans le classeur actif, par exemple copier la « feuille 7 » dans un nouveau fichier, que l'on nommera « *Démo1* » dans « *C:\Formation_VBA* » :

```
Sheets("Feuil7").Copy
ActiveWorkbook.Close SaveChanges:=True, Filename:="C:\Formation_VBA\Démo1"
```

C'est moins simple pour copier cette feuille dans un fichier existant, nommé « *Démo2* », comme première feuille du classeur. Car l'ouverture du classeur « *Démo2* » le rend actif. Il faut donc revenir sur le classeur d'origine pour copier la feuille d'origine « Feuil7 ». Sans cette manipulation, vous vous trompez de classeur et donc de feuille.

Et si plusieurs classeurs sont ouverts, vous risquez de vous perdre. La meilleure solution est alors de travailler avec des objets qui représentent les classeurs concernés : un objet *WbSource* représente le classeur source, un autre WbDest représente le classeur destination :

```
Dim WbSource As Workbook ' Objet Workbook pour le classeur source.
Dim WbDest As Workbook   ' Objet Workbook pour le classeur destination.

Set WbSource = ActiveWorkbook                              ' Mémorise le classeur actif =
 Source.
Set WbDest = Workbooks.Open("C:\Formation_VBA\Démo2.xlsx")  ' Ouvre le fichier destination.
WbSource.Sheets("Feuil7").Copy Before:=WbDest.Sheets(1)     ' Copie Feuille 7 vers ce fichier.
WbDest.Close SaveChanges:=True                             ' Ferme Démo2.xlsx en sauvegardant.
WbSource.Activate
```

Dans la même logique, il est possible de copier une partie de la « feuille 7 », par exemple une plage des colonnes « B à G », vers la feuille « *Données* » dans la cellule « A1 » du fichier existant « Démo2 ».

ThisWorkbook représente le classeur qui contient le code VBA :

```
Dim WbSource As Workbook ' Objet Workbook pour le classeur source.
Dim WbDest As Workbook   ' Objet Workbook pour le classeur destination.

Set WbSource = ThisWorkbook                                ' Classeur qui contient le code.
Set WbDest = Workbooks.Open("C:\Formation_VBA\Démo2.xlsx") ' Ouvre le fichier destination.
WbSource.Sheets("Feuil7").Range("B:G").Copy _
                          Destination:=WbDest.Sheets("Données").Range("A1") ' Copie...
WbDest.Close SaveChanges:=True                             ' Ferme Démo2.xlsx en sauvegardant.
WbSource.Activate                                          ' Active le fichier source.
```

Vous pouvez travailler sur des classeurs différents de celui qui contient le code source :

```
Dim WbSource As Workbook ' Objet Workbook pour le classeur source.
Dim WbDest As Workbook   ' Objet Workbook pour le classeur destination.

Set WbSource = Workbooks.Open("C:\Formation_VBA\Démo2") ' Ouvre le fichier Source.
Set WbDest = Workbooks.Open("C:\Formation_VBA\Démo3")   ' Ouvre le fichier Destination.

' Boucle sur les données de la source tant qu'il y a des lignes.
ys = 2: yD = 1
While WbSource.Sheets("Test").Cells(ys, 1) <> ""
    ' Si la couleur préférée est rouge alors recopie la civilité.
    If WbSource.Sheets("Test").Cells(ys, 7) = "rouge" Then
        WbDest.Sheets("Feuil1").Cells(yD, 1) = _
        WbSource.Sheets("Test").Cells(ys, 2)
        yD = yD + 1 ' Incrémente la ligne destination.
    End If
    ys = ys + 1 ' Passe à la ligne source suivante.
Wend

WbSource.Close SaveChanges:=False ' Ferme la source sans sauvegarder.
WbDest.Close SaveChanges:=True    ' Ferme la destination en sauvegardant.
```

Ici le traitement est très rapide, car il y a peu de données à gérer. Mais il en va autrement pour les traitements plus importants, sur des feuilles contenant des formules. Il faut alors optimiser les traitements, tout en faisant patienter l'utilisateur.

Astuces à appliquer pour accélérer les traitements importants :

- avant le lancement du traitement :
    - affichez un message dans la barre d'état : `Application.StatusBar = "Merci de patienter…"`,
    - empêchez l'utilisateur d'interrompre le traitement : `Application.EnableCancelKey = xlDisabled`,
    - mettez le curseur de la souris en forme de sablier : `Application.Cursor = xlWait`,
    - désactivez la mise à jour de l'écran : `Application.ScreenUpdating = False`,
    - désactivez le calcul automatique : `Application.Calculation = xlCalculationManual` (après avoir pris soin de sauvegarder le statut d'origine dans une variable : `XlCalculation = Application.Calculation`),
    - désactivez les événements : `Application.EnableEvents = False` ;
- quand tout est terminé, ou en cas d'erreur dans le traitement, rétablissez la situation d'origine :
    - calculs automatiques ou manuels : `Application.Calculation =` statut d'origine sauvegardé précédemment,
    - mise à jour de l'écran : `Application.ScreenUpdating = True`,
    - curseur en forme classique : `Application.Cursor = xlDefault`,
    - touche [Echap] de nouveau disponible : `Application.EnableCancelKey = xlInterrupt`,
    - restaurez la barre d'état : `Application.StatusBar = False`,
    - restaurez les événements : `Application.EnableEvents = True`.

Astuces pour des traitements plus souples

Jusqu'à présent nous avons utilisé **Range(Adresse)** ou **Cells(Ligne, Colonne)** pour atteindre une cellule.

L'inconvénient de ces méthodes est que les cellules doivent être figées. Il suffit que l'utilisateur insère une colonne ou une ligne au mauvais endroit pour que notre code pointe sur de mauvaises références.

Pour y remédier, nous allons donner un nom à la cellule désirée, par exemple en « A1 », via le menu « Formules », « Définir un nom ». Donnez le nom « MaRef » et choisissez « Classeur » comme zone.
Peu importe maintenant si l'utilisateur déplace cette cellule, car votre code fait référence à une cellule nommée et EXCEL se charge de la trouver. Et en définissant comme zone (étendue) le « Classeur », vous n'avez même pas besoin d'indiquer le nom de la feuille où se trouve la référence.

Vous modifiez la valeur de cette cellule par l'instruction : `Range("MaRef").value = 15`.
Pour connaître sa ligne : `y = Range("MaRef").Row`.
Pour connaître sa colonne : `x = Range("MaRef").Column`.

De même pour remplacer dans votre code les données écrites en dur, utilisez une feuille, où seront écrites ces informations. Pour masquer la feuille 1 : `Worksheets("Feuil1").Visible = False` ou `xlVeryHidden` pour qu'elle ne puisse pas être affichée par l'utilisateur.

La plage d'un tableau peut aussi être nommée. C'est pratique sur un tableau figé, mais ça devient contre-productif sur un tableau évolutif. Nous allons étudier comment manipuler les données d'un tableau…

XX - Manipuler les données d'un tableau

Manipuler les données d'un tableau est souvent délicat, car le nombre de lignes, et plus rarement de colonnes, peut varier en fonction des ajouts ou des suppressions des données.

La grande difficulté est alors de connaître la dimension exacte du tableau.
Le premier bon réflexe est d'attribuer une ligne d'en-tête à vos tableaux. Elle nous sera utile par la suite.
Le deuxième est de nommer la première cellule de l'en-tête, soit le point d'origine du tableau, et non pas toute la zone du tableau.

Dans cet exemple, la cellule « B2 » est nommée MonTableau :

	A	B	C	D	E	F	G
2		Civilité	Anglais	Espagnol	Allemand	Autre	Couleur
3		Femme	VRAI	VRAI	FAUX	FAUX	rouge
4		Femme	VRAI	FAUX	VRAI	FAUX	rouge
5		Femme	FAUX	VRAI	FAUX	VRAI	rouge
6		Homme	VRAI	FAUX	VRAI	FAUX	gris
7		Femme	FAUX	VRAI	FAUX	VRAI	rouge
8		Homme	VRAI	FAUX	VRAI	FAUX	gris

Nous savons déterminer l'origine du tableau :

```
OrigineY = Range("MonTableau").Row
OrigineX = Range("MonTableau").Column
```

Nous pourrions déterminer la fin du tableau avec :

```
FinY = Range("MonTableau").SpecialCells(xlCellTypeLastCell).Row
FinX = Range("MonTableau").SpecialCells(xlCellTypeLastCell).Column
```

Mais c'est un piège, car ne sont prises en compte que les données visibles, et si vous mettez un filtre sur la « Civilité » en sélectionnant « Femme », le tableau se termine en ligne 7, et c'est cette valeur qui est renvoyée au lieu de la ligne 8.

Nous allons procéder autrement, après avoir sélectionné MonTableau, en utilisant la propriété **CurrentRegion** de l'objet **Range** qui étendent automatiquement la sélection de façon à y inclure toute la zone en cours, y compris les lignes masquées :

sélection de l'origine du tableau : Range("MonTableau").Select ;
étendre la sélection à toute la zone : ActiveCell.CurrentRegion.Select ;

en bouclant sur les cellules de la zone ainsi sélectionnée Areas(1).Cells, il est possible de retrouver la dernière ligne FinY et la dernière colonne FinX, et donc les coordonnées maximales du tableau :

```
For Each Cellule In Selection.Areas(1).Cells
    If FinY < Cellule.Row Then FinY = Cellule.Row
    If FinX < Cellule.Column Then FinX = Cellule.Column
Next Cellule
```

Nous pouvons obtenir la zone des données, donc sans l'en-tête du tableau, en définissant un objet **Range** qui s'étend des coordonnées (OrigineY+1, OrigineX) à (FinY, FinX). En effet **Range** peut représenter aussi une plage de cellules, en utilisant le format : Range(Cellule1, Cellule2).

Vous remarquez que l'indice n de Areas(n) est 1, car il n'y a qu'une seule zone. Par contre il y a plusieurs cellules dans cette zone. Les cellules peuvent aussi être atteintes individuellement par l'instruction Selection.Areas(1).cells(i).Propriété dans une boucle de type For i = 1 To Selection.Areas(1).Count.

Ne confondez pas :

- Selection.Areas(n).Count, qui donne le nombre de cellules sélectionnées dans la zone n ;
- Selection.Areas.Count, qui donne le nombre de zones.

Pour connaître le nombre total de cellules sélectionnées, toutes zones confondues, il faut soit faire une boucle sur les zones :

```
 For i = 1 To Selection.Areas.Count
    NbCelSélectionnées = NbCelSélectionnées + Selection.Areas(i).Count
Next i
```

soit utiliser l'instruction : NbCelSélectionnées = Selection.Count.

Gardez à l'esprit ce concept, car vous aurez besoin, pour certains traitements, de vous assurer qu'une seule cellule est sélectionnée, ou inversement, vous devrez atteindre toutes les cellules sélectionnées, même si elles sont réparties sur plusieurs zones.

Pour atteindre toutes les cellules sélectionnées, utilisez une boucle de ce type :

```
 Dim Cellule As Range
For Each Cellule In Selection
    Cellule.Interior.Color = 65535
Next Cellule
```

La fonction suivante utilise en argument le nom du tableau à analyser, et quatre variables de type **ByRef**, par défaut, pour les coordonnées du tableau, qui seront ainsi alimentées, et retourne dans un objet **Range** la zone contenant les données (ou **Nothing** si une erreur est rencontrée) :

```
'-------------------------------------------------------------------------------
Function TableauCoordonnées(ByVal StrTableau As String, Optional OrigineY As Long, _
   Optional OrigineX As Long, Optional FinY As Long, Optional FinX As Long) As Range
'-------------------------------------------------------------------------------
Dim Cellule As Range  ' Variable qui représente les cellules de la zone sélectionnée.
FinY = 0: FinX = 0    ' Efface les valeurs par sécurité.

On Error GoTo Err_CT                ' Gestion des erreurs.
Range(StrTableau).Select            ' Sélectionne le tableau.
OrigineY = Range(StrTableau).Row    ' Détecte l'origine Y
OrigineX = Range(StrTableau).Column ' Détecte l'origine X
ActiveCell.CurrentRegion.Select     ' Étend la sélection à toute la zone.

For Each Cellule In Selection.Areas(1).Cells ' Boucle sur les cellules de la zone.
    If FinY < Cellule.Row Then FinY = Cellule.Row         ' Ligne maximale.
    If FinX < Cellule.Column Then FinX = Cellule.Column   ' Colonne maximale.
Next Cellule                                              ' Cellule suivante.

' Retourne la zone du tableau sans l'en-tête, pour des traitements ultérieurs :
Set TableauCoordonnées = Range(Cells(OrigineY + 1, OrigineX), Cells(FinY, FinX))
Range(StrTableau).Select          ' Sélectionne le tableau uniquement.
Err_CT:
End Function
```

Dans l'exemple ci-dessous, nous appelons cette fonction, qui retourne dans la variable Données la zone contenant les données. Une boucle sur ces données met en fond jaune les cellules qui valent « Faux » :

```
'-------------------------------------------------------------------------------
Sub AnalyseMonTableau()
'-------------------------------------------------------------------------------
Dim Données As Range ' Variable qui représente la zone de données.
Dim Cellule As Range ' Variable qui représente une cellule de cette zone.
Dim y As Long, x As Long, YY As Long, XX As Long ' Coordonnées du tableau.

' Recherche les coordonnées du tableau, et retourne la zone de données :
Set Données = VBO.TableauCoordonnées("MonTableau", y, x, YY, XX)
If Données Is Nothing = True Then MsgBox "Erreur": Exit Sub ' Quitte si erreur

For Each Cellule In Données ' Boucle sur les données du tableau.
    Cellule.Interior.Color = xlNone ' Efface la couleur de fond.
    ' Met le fond en jaune si la cellule vaut False.
    If Cellule.Value = False Then Cellule.Interior.Color = 65535
Next Cellule                ' Cellule suivante de la zone de données.

End Sub
```

La prochaine ligne du tableau où insérer de nouvelles données est en YY+1.
Si une erreur se produit alors la variable Données vaut « Rien » et la procédure est quittée.

La boucle For Each Cellule In Données… Next Cellule remplace les boucles :

```
 For Ligne = Y + 1 To YY
    For Colonne = X To XX
        ...
    Next Colonne
Next Ligne
```

Si les coordonnées du tableau ne sont pas utiles, et que vous souhaitez uniquement sélectionner les données, l'appel à la fonction peut être simplifié par :

```
Set Données = VBO.TableauCoordonnées("MonTableau")
```

Inversement, si la zone des données n'est pas utile, et que vous souhaitez uniquement connaître les coordonnées du tableau, l'appel à la fonction peut être simplifié par :

```
Call VBO.TableauCoordonnées("MonTableau", Y, X, YY, XX)
```

Pour trier le tableau j'ai enregistré une macro du tri réalisé manuellement, puis j'ai étudié le code généré :

```
Range("B2").Select
ActiveWorkBook.WorkSheets("Feuil7").Sort.SortFields.Clear
ActiveWorkBook.WorkSheets("Feuil7").Sort.SortFields.Add Key:=Range("B3:B8"), _
    SortOn:=xlSortOnValues Order:=xlAscending, DataOption:=xlSortNormal
With ActiveWorkBook.WorkSheets("Feuil7").Sort
    .SetRange Range("B2:G8")
    .Header = xlYes
    .MatchCase = False
    .Orientation = xlTopToBottom
    .SortMethod = xlPinYin
    .Apply
End With
```

J'ai éliminé les parties du code qui reprennent les options par défaut, car elles n'apportent rien de pertinent.

J'ai remplacé les données fixes par des données variables pour pouvoir faire une fonction réutilisable dans tous les tableaux, que j'ajouterai à ma boîte à outils :

- Range("B2").Select sera remplacé par les coordonnées du tableau à trier ;
- Key:=Range("B3:B8") sera remplacé par l'adresse des données de la colonne à trier ;
- ActiveWorkbook.Worksheets("Feuil7") sera remplacé par la feuille active ;
- SetRange Range("B2:G8") sera remplacé par la zone de données.

Dans cette fonction, je passe en argument dans StrTableau la cellule nommée qui représente l'origine du tableau à trier, dans TrierPar le libellé de l'en-tête de la colonne à trier, et trois arguments facultatifs concernant la méthode de tri, basés sur les énumérations EXCEL, à savoir TrierSur (par défaut trie les valeurs, il est possible de trier par couleur de cellule ou de police), TrierOrdre (par défaut trie par ordre croissant) et TrierOption (par défaut trie les données numériques et textuelles séparément).

```
'-------------------------------------------------------------------------------
Function TableauTrier(ByVal StrTableau As String, TrierPar As String, _
            Optional TrierSur As XlSortOn = xlSortOnValues, _
            Optional TrierOrdre As XlSortOrder = xlAscending, _
            Optional TrierOption As XlSortDataOption = xlSortNormal) As Boolean
'-------------------------------------------------------------------------------
Dim Données As Range ' Variable qui représente la zone de données.
Dim y As Long, x As Long, YY As Long, XX As Long ' Coordonnées du tableau.
Dim i As Long
On Error GoTo Err_Tri  ' Gestion des erreurs.

' Recherche les coordonnées du tableau, et retourne la zone de données :
Set Données = TableauCoordonnées(StrTableau, y, x, YY, XX)

' Boucle sur les en-têtes pour trouver la colonne Trier Par :
For i = x To XX
    If Cells(y, i) = TrierPar Then Exit For
Next i

With ActiveSheet.Sort ' Configuration du tri :
    .SortFields.Clear ' Efface les anciens critères.          Colonne du tri...
    .SortFields.Add Key:=Range(Cells(y + 1, i), Cells(YY, i)), _
     SortOn:=TrierSur, Order:=TrierOrdre, DataOption:=TrierOption ' et options.
    .SetRange Données ' Plage de tri = zone de données.
    .Header = xlNo    ' Pas d'en-tête.
    .Apply            ' Exécute le tri.
End With
TableauTrier = True         ' Retourne Vrai si le tri est effectué.
Range(StrTableau).Select    ' Sélectionne l'origine du tableau.
Err_Tri:                    ' Retourne Faux en cas d'erreur.
End Function
```

Si le tri se fait correctement, la fonction retourne « Vrai ».

Exemples d'appels de la fonction :

```
Call VBO.TableauTrier("MonTableau", "Civilité")
Call VBO.TableauTrier("MonTableau", "Civilité", xlSortOnValues, xlDescending, xlSortNormal)
```

En manipulant les données des tableaux, vous serez amené à insérer ou supprimer, des lignes ou des colonnes. Les instructions en VBA sont les suivantes :

- Rows renvoie un objet qui représente les lignes d'une feuille de calcul.

La méthode **Delete** permet de supprimer une ligne, la méthode **Insert** permet d'insérer une ligne ;

- Rows(6).Delete supprime la ligne 6 de la feuille active. Ou écrivez Rows(y).Delete si y vaut 6 ;
- Rows("2:5").Delete supprime les lignes 2 à 5 de la feuille active.

Si vous utilisez des variables y1 et y2 qui valent respectivement 2 et 5, la syntaxe est :

```
Rows("" & y1 & ":" & y2 & "").Delete>
```

Et donc pour insérer 4 lignes après la ligne 1 : Rows("2:5").Insert

De même **Columns** renvoie un objet qui représente les colonnes d'une feuille de calcul ;

- Columns(6).Delete supprime la colonne « F » de la feuille active. Vous pouvez aussi écrire Columns("F").Delete, ou bien Columns(x).Delete si x vaut 6 ;
- Columns ("B:E").Delete supprime les colonnes « B » à « E » de la feuille active.

Si vous utilisez des variables x1 et x2 qui valent respectivement 2 et 5, vous devez transformer les variables en références alphabétiques, ce qui complique l'exercice. Le plus simple est de répéter n fois la suppression de la colonne x1, où n = x2 - x1 + 1 ;

- Columns(6).Insert permet d'insérer une colonne en « F ».

Vous serez aussi amené à gérer les cellules sélectionnées. La fonction ci-dessous permet de ne retenir dans une sélection que les cellules qui répondent à un critère, pour pouvoir agir dessus après, avec en option la possibilité d'échanger la valeur de ces cellules. Voici un exemple d'appel pour sélectionner les données d'un tableau, puis remplacer celles qui valent « rouge » par « (rouge) », puis mettre ces cellules en fond rouge si effectivement des données répondent à ce critère :

```
Dim Données As Range ' Variable qui représente la zone de données.
Set Données = VBO.TableauCoordonnées("A1") ' Détermine la plage du tableau d'origine A1
Données.Select ' Sélectionne les données de cette plage pour pouvoir faire un échange.
If VBO.CellulesActivesSélectionnerRéférence("rouge", True, "(rouge") > 0 Then
    Selection.Interior.Color = 255 ' Met le fond de ces cellules en rouge.
End If

'-------------------------------------------------------------------------------
Function CellulesActivesSélectionnerRéférence(Référence As String, _
                Optional EgaleA As Boolean = True, Optional RemplacerPar As Variant) As Long
'-------------------------------------------------------------------------------
Dim Cellule As Range            ' Représente les cellules sélectionnées.
Dim Def_Plage As Range          ' Plage qui représentera les cellules sélectionnées.
For Each Cellule In Selection   ' Boucle sur les cellules sélectionnées :
    If (Cellule.Value = Référence And EgaleA = True) _
    Or (Cellule.Value <> Référence And EgaleA = False) _
    Then ' Si la cellule correspond aux critères de comparaison.
        If Def_Plage Is Nothing = True Then
```

```
            If IsMissing(RemplacerPar) = False Then Cellule.Value = RemplacerPar  ' Remplacement.
            Set Def_Plage = Union(Range(Cellule.Address), Range(Cellule.Address)) ' 1re cellule.
        Else ' Sinon complète la solution.
            If IsMissing(RemplacerPar) = False Then Cellule.Value = RemplacerPar  ' Remplacement.
            Set Def_Plage = Union(Def_Plage, Range(Cellule.Address))              ' Plage.
        End If
    End If
Next Cellule
On Error Resume Next
Def_Plage.Select                                        ' Sélectionne les cellules.
CellulesActivesSélectionnerRéférence = 0                ' Retourne 0 par défaut.
CellulesActivesSélectionnerRéférence = Def_Plage.Count  ' ou le nombre de sélections.
End Function
```

Je souhaite maintenant indiquer à la suite du tableau, dans la colonne « Civilité », le nombre de femmes et sur la ligne suivante le nombre d'hommes concernés par le sondage.

La première solution est de faire une boucle sur les données de la colonne et de compter le nombre d'éléments concernés.
Deuxième solution, nous allons utiliser en VBA la formule EXCEL qui compte le nombre de cellules d'une plage qui répondent à une condition.

La formule EXCEL pour compter le nombre de cellules valant « Femme » dans la plage « B2:B8 » est :

```
=NB.SI(B2:B8;"Femme")
```

Notre tableau n'étant pas figé, nous allons utiliser des références relatives, en nous servant des coordonnées du tableau, Y, X, YY, XX.

La formule n'acceptant que des adresses au format « A1 » il nous faut auparavant convertir Range(Cells(2, 2), Cells(8, 2)) en « **B2:B8** », grâce à la propriété **Adress** de l'objet **Range**.

```
Adresse = Range(Cells(Y, X), Cells(YY, X)).Address(False, False, xlA1, False)
```

La formule sera appliquée à la cellule désirée par la propriété **FormulaLocal** de l'objet **Range**. Ou utilisez la propriété **Formula** de cet objet pour passer la formule en anglais :

```
Cells(YY + 2, X).FormulaLocal = "=NB.SI(" & Adresse & ";""Femme"")"
```

Remarquez l'usage des deux doubles guillemets pour passer du texte en argument et non pas une valeur.

Comme nous l'avons déjà vu, le signe « & » permet de concaténer la variable Adresse dans la formule.

```
'-------------------------------------------------------------------------------
Sub AnalyseMonTableauBIS()
'-------------------------------------------------------------------------------
Dim y As Long, x As Long, YY As Long, XX As Long ' Coordonnées du tableau.
Dim Adresse As String ' Variable qui contiendra l'adresse d'une plage.

Call VBO.TableauCoordonnées("MonTableau", y, x, YY, XX) ' Alimente les coordonnées.

Adresse = Range(Cells(y, x), Cells(YY, x)).Address(False, False, xlA1, False)
Cells(YY + 2, x).FormulaLocal = "=NB.SI(" & Adresse & ";""Femme"")" ' Nb de femmes.
Cells(YY + 3, x).FormulaLocal = "=NB.SI(" & Adresse & ";""Homme"")" ' Nb d'hommes.

End Sub
```

De même nous pouvons utiliser une formule pour compter le nombre de personnes ayant étudié l'anglais.

```
Adresse = Range(Cells(Y, X + 1), Cells(YY, X + 1)).Address(False, False, xlA1, False)
Cells(YY + 3, X + 1).FormulaLocal = "=NB.SI(" & Adresse & ";""VRAI"")"
```

Cette formule utilise une plage relative au format « **B2:B8** », et non pas figée au format « **$B$2:$B$8** ». Elle peut donc être réutilisée dans les colonnes suivantes, pour compter le nombre de personnes ayant étudié l'espagnol, l'allemand, une autre langue, ce qui économise le calcul de l'adresse de la colonne :

```
Cells(YY + 3, X + 2).FormulaR1C1Local = Cells(YY + 3, X + 1).FormulaR1C1Local
Cells(YY + 3, X + 3).FormulaR1C1Local = Cells(YY + 3, X + 1).FormulaR1C1Local
Cells(YY + 3, X + 4).FormulaR1C1Local = Cells(YY + 3, X + 1).FormulaR1C1Local
```

Ne confondez pas **FormulaLocal** qui stocke la formule au format « **B2:B8** » et **FormulaR1C1Local** qui stocke au format « (Ligne - Y, Colonne - X) : (Ligne - Y, Colonne - X) ».

Nous pouvons nommer une cellule en VBA, et réutiliser ce nom pour désigner la cellule concernée, avec la méthode **Add** de l'objet **Names,** en renseignant les arguments suivants :

- Name : nom d'identification ;
- RefertsTo : adresse de la cellule, ou de la plage de cellules.

Consultez l'aide pour plus d'informations sur les différents arguments.

Par exemple :

```
ActiveWorkbook.Names.Add Name:="Nom", RefersTo:="=" & Cells(Y, X).Address
```

Comme les arguments de la fonction sont dans l'ordre, vous pouvez utiliser cette autre syntaxe :

```
ActiveWorkbook.Names.Add "Nom", "=" & Cells(Y, X).Address
```

Pour supprimer un nom, il faut utiliser la méthode **Delete** de l'objet :

```
ActiveWorkbook.Names("Nom").Delete
```

Dans l'exemple qui suit, la formule pour calculer la colonne « Anglais » est attribuée à la cellule(YY+3,X+1). Cette cellule est nommée « Test ». Elle est réutilisée pour les trois colonnes suivantes. Puis le nom est supprimé.

```
Adresse = Range(Cells(y, x + 1), Cells(YY, x + 1)).Address(False, False, xlA1, False)
Cells(YY + 3, x + 1).FormulaLocal = "=NB.SI(" & Adresse & ";""VRAI"")" ' Nb Anglais

' Nomme la cellule pour pouvoir la réutiliser :
ActiveWorkbook.Names.Add Name:="Test", RefersTo:="=" & Cells(YY + 3, x + 1).Address

Cells(YY + 3, x + 2).FormulaR1C1Local = Range("Test").FormulaR1C1Local ' Nb Espagnol
Cells(YY + 3, x + 3).FormulaR1C1Local = Range("Test").FormulaR1C1Local ' Nb Allemand
Cells(YY + 3, x + 4).FormulaR1C1Local = Range("Test").FormulaR1C1Local ' Nb Autre

ActiveWorkbook.Names("Test").Delete ' Supprime le nom qui ne sert plus.
```

Peut-être préférez-vous avoir la valeur calculée plutôt que la formule.

Dans ce cas, il faut remplacer la formule par la valeur :

```
Cells(YY + 3, X + 2).Formula = Cells(YY + 3, X + 2).Value
```

L'utilisation des formules d'EXCEL peut rendre de grands services dans vos développements VBA et remplacer avantageusement de nombreuses lignes de codes.

Cet exemple utilise la formule **=RECHERCHEV** pour rechercher la couleur préférée du premier homme trouvé dans un tableau, où la civilité est en colonne B et la couleur préférée en colonne G :

```
ActiveCell.Formula = "=VLOOKUP(""Homme"", B:G,6,FALSE)"
If IsError(ActiveCell) = False Then MaRecherche = ActiveCell
ActiveCell.Formula = ""
```

Cet exemple utilise la formule **=EQUIV** pour trouver la ligne où le texte « bleu » est dans la colonne G, ou 0 si ce texte n'est pas trouvé :

```
Range("A1").Formula = "=MATCH(""bleu"", G:G,0)"
If IsError(Range("A1")) = False Then Range("A1") = 0
Y = Range("A1")
Range("A1") = ""
```

XXI - Les procédures récursives

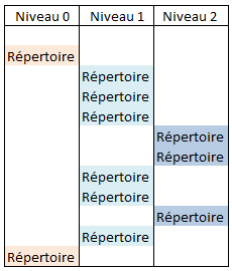
Une procédure est récursive quand elle s'appelle elle-même. Vous en connaissez déjà une : QuickSort.

Nous allons en créer une sur un cas d'école plus simple à comprendre : l'analyse des répertoires et sous-répertoires d'un disque.

Où il faut boucler sur les répertoires du niveau N, et pour chaque sous-répertoire trouvé, boucler sur les répertoires du niveau N+1, et pour chaque sous-répertoire trouvé, boucler sur les répertoires du niveau N+2. Revenir au niveau N+1 quand le niveau N+2 a fini d'être analysé. Et ainsi de suite.

C'est la même fonction qui est utilisée à chaque fois, en passant en argument le répertoire du niveau.

C'est pourquoi elle est récursive.



Au lieu d'utiliser la fonction **Dir**, je vais utiliser un objet ActiveX plus puissant, **Scripting.FileSystemObject** (système de fichiers de Windows), mais qui nécessite une explication :

- il faut en premier lieu créer un objet, ici ObjFso, qui fait référence à l'objet ActiveX ;
- puis créer un objet, ici ObjDossier, qui fait référence à la méthode **GetFolder** de l'objet ObjFso. Cet objet contiendra la liste des dossiers (ou répertoires) ;
- enfin créer un objet, ici ObjSousRep, qui fait référence à la méthode **SubFolders** de l'objet ObjDossier. Cet objet contiendra la liste des sous-répertoires.

  Il y aura une boucle sur les sous-répertoires, et un appel récursif pour chacun d'eux. Appel qui permettra d'analyser ses sous-répertoires.

J'affiche en colonne 1 de la feuille active les répertoires trouvés.

Cette fonction peut être appelée depuis la fenêtre d'exécution par : MesRepertoires("P:")

```
'-------------------------------------------------------------------------
Sub MesRepertoires(StrRepertoire As String)
'-------------------------------------------------------------------------
Dim ObjFso, ObjDossier, ObjSousRep, SousRep

Set ObjFso = CreateObject("Scripting.FileSystemObject") ' Objet ActiveX
Set ObjDossier = ObjFso.GetFolder(StrRepertoire)        ' Objet Dossier.
Set ObjSousRep = ObjDossier.SubFolders                  ' Objet Sous-répertoire.
Static y As Long
```

```
For Each SousRep In ObjSousRep ' Boucle sur les sous-répertoires du dossier.

    If SousRep.Attributes = 16 Or SousRep.Attributes = 17 Then ' Si attribut Normal.

        Cells(y + 1, 1) = SousRep.Path ' Affiche le sous-répertoire
        y = y + 1

        Call MesRepertoires(SousRep.Path) ' Récursivité sur le sous-répertoire.

Next SousRep ' Sous-répertoire suivant.

End Sub
```

Pour analyser un disque dur, il est préférable de faire l'impasse sur les dossiers système, qui sont parfois protégés en accès et génèrent des erreurs en cas de tentative de lecture.
La propriété **Attributes** de l'objet **SubFolders** vaut 16 ou 17 pour les dossiers ordinaires.

En pratique, c'est plus une procédure de recherche d'un fichier en incluant les sous-répertoires qui vous sera utile. C'est ce que je vous propose ici.
J'ai ajouté la possibilité de mémoriser les résultats trouvés dans un tableau de données, et la possibilité de cumuler ou non les résultats de plusieurs recherches.
Pour la comparaison des fichiers analysés avec le fichier recherché, j'utilise l'instruction **Like** qui permet d'utiliser le point d'interrogation comme joker sur un caractère, et l'étoile comme joker sur une suite de caractères. Ainsi la recherche au format « *.xl* » correspond à rechercher tous les fichiers EXCEL.

```
'-------------------------------------------------------------------------------
Function RechercheFichier(ByVal StrRepertoire As String, ByVal StrFichier As String, _
                          ByRef TableauAnalyse() As Variant, _
                          Optional Initialisation As Boolean = True) As Long
'-------------------------------------------------------------------------------
Dim ObjFso, ObjDossier, ObjSousRep, SousRep, Fichier, TestErr
Static NbAnalyse As Long                                ' Nbre de fichiers trouvés

If Initialisation = True Then NbAnalyse = 0             ' Réinitialise le tableau.

Set ObjFso = CreateObject("Scripting.FileSystemObject") ' Objet ActiveX
Set ObjDossier = ObjFso.GetFolder(StrRepertoire)        ' Objet Dossier.
Set ObjSousRep = ObjDossier.SubFolders                  ' Objet Sous-répertoire.

On Error GoTo FinProcedure ' Gestion des erreurs.

' Boucle sur les fichiers du dossier :
For Each Fichier In ObjDossier.Files

    ' Si ce fichier correspond à la recherche alors le mémoriser :
    If Fichier.Name Like StrFichier = True Then
        ReDim Preserve TableauAnalyse(NbAnalyse)
        TableauAnalyse(NbAnalyse) = Fichier.Path
        NbAnalyse = NbAnalyse + 1
    End If

Next Fichier

' Boucle sur les sous-répertoires du dossier hors répertoires système :
For Each SousRep In ObjSousRep
    If SousRep.Attributes = 16 Or SousRep.Attributes = 17 Then
        TestErr = Dir(SousRep.Path & "\*.*") ' Test pour les dossiers protégés.
        Call RechercheFichier(SousRep.Path, StrFichier, TableauAnalyse(), False)
    End If
Next SousRep

FinProcedure:
RechercheFichier = NbAnalyse
End Function
```

Exemples d'appels de la fonction :

- pour stocker dans le tableau de données Tableau() l'adresse de tous les fichiers EXCEL, dans le répertoire de l'application, en incluant les éventuels sous-répertoires :

```
Call VBO.RechercheFichier(ThisWorkbook.Path, "*.xl*", Tableau())
```

- pour stocker dans le tableau de données Tableau() l'adresse du fichier « Sapi.xls », dans le répertoire de l'utilisateur, en incluant les éventuels sous-répertoires :

```
Call VBO.RechercheFichier(Application.DefaultFilePath, "Sapi.xls", Tableau())
```

- pour ajouter au tableau de données Tableau() l'adresse des fichiers contenant « Sapi » dans leur nom, dans le répertoire de l'utilisateur, en incluant les éventuels sous-répertoires :

```
Call VBO.RechercheFichier(Application.DefaultFilePath, "*Sapi*", Tableau(), False)
```

Dans l'exemple suivant, je recherche le fichier EXCEL « Sapi » dans le répertoire de l'application, puis dans le répertoire de l'utilisateur, puis dans « P: ».

Si un seul fichier est trouvé, je peux récupérer son adresse, c'est l'élément d'indice 0 du tableau, mais si plusieurs fichiers sont trouvés, il faut que l'utilisateur sélectionne celui qu'il désire. Pour cela j'ai créé un formulaire, nommé « Liste_Choix » qui contient une zone de liste « ComboBox1 » et un bouton de commande « CommandButton1 ». Ce bouton a un événement « Sur click » qui mémorise le choix fait, dans la propriété **Tag** (commentaire) du formulaire avant de refermer le formulaire.

```
Private Sub CommandButton1_Click()
Liste_Choix.Tag = ComboBox1.Value
Liste_Choix.Hide
End Sub
'---------------------------------------------------------------------------
Function SelectionneSapi() As String
'---------------------------------------------------------------------------
Dim i As Long, Tableau() As Variant

' Recherche le fichier SAPI.XLS dans le répertoire de l'application,
' puis de l'utilisateur, puis sur P, et cumule les résultats :
Call VBO.RechercheFichier(ThisWorkbook.Path, "Sapi.xl*", Tableau())
Call VBO.RechercheFichier(Application.DefaultFilePath, "Sapi*.xl*", Tableau(), False)
Call VBO.RechercheFichier("P:\", "Sapi.xl*", Tableau(), False)

' Si un seul résultat trouvé alors le prendre :
If UBound(Tableau) = 0 Then SelectionneSapi = Tableau(0): GoTo Fin

Call VBO.TriRapide(Tableau())     ' Trie les résultats par ordre croissant.
Liste_Choix.ComboBox1.Clear       ' Efface toutes les anciennes données de la liste
For i = 0 To UBound(Tableau())    ' Boucle sur le tableau et...
    Liste_Choix.ComboBox1.AddItem Tableau(i) ' Alimente la liste déroulante.
Next i

Liste_Choix.Show                  ' Affiche la liste déroulante.
SelectionneSapi = Liste_Choix.Tag ' Récupère le choix fait.
Unload Liste_Choix                ' Libère le formulaire de la mémoire.
Fin:
End Function
```

Le fait de stocker dans un tableau les solutions possibles peut être intéressant, mais dans la majorité des cas, cette fonction sera utilisée pour rechercher la présence d'un fichier spécifique dans un répertoire connu, en étendant la recherche aux sous-répertoires. Nous avons développé notre procédure, car VBA ne possède pas de procédure pour faire cela.

Sauf… que VBA permet d'incorporer des procédures externes : ce sont les API.

XXII - Les API

> « Les API, Application Programming Interface, sont des ensembles de classes, de méthodes ou de fonctions, mises gratuitement à disposition des développeurs par les éditeurs de services, de logiciels ou d'applications. Les développeurs en feront usage pour intégrer certaines fonctionnalités à leurs applications. »

Les API compatibles avec VBA sont nombreuses. Vous trouverez des informations sur internet.

Mon site préféré : **http://allapi.mentalis.org/apilist/apilist.php**

Attention cependant avec l'utilisation des API, car si elles sont souvent bien plus rapides à l'exécution que les fonctions écrites en VBA, c'est parce qu'elles ont la particularité d'exploiter directement les ressources du système d'exploitation. En contrepartie, le système d'exploitation utilisé peut avoir une incidence sur leur comportement, et rien ne vous assure que les développements réalisés aujourd'hui avec des API seront compatibles avec la prochaine version de Windows.

Les API doivent être déclarées en en-tête d'un module. C'est une syntaxe très proche de celle utilisée pour nos procédures. Ici nous allons intégrer la procédure **SearchTreeForFile** qui est contenue dans le fichier (la bibliothèque) « imagehlp.dll » :

```
Declare Function SearchTreeForFile Lib "imagehlp" (ByVal RootPath As String, ByVal
 InputPathName As String, ByVal OutputPathBuffer As String) As Long
```

Désormais, notre application connaît une nouvelle procédure permettant de rechercher un fichier dans un dossier et ses sous-dossiers, avec les arguments suivants :

- RootPath : est le répertoire où commencer la recherche ;
- InputPathName : est le nom du fichier recherché. L'argument accepte le point d'interrogation et l'étoile comme jokers ;
- OutputPathBuffer : est une variable temporaire qui contiendra le résultat de la recherche.

La fonction retourne 0 si aucun fichier n'est trouvé ou un chiffre différent de 0 dans le cas contraire, mais ne retourne pas directement le chemin trouvé. Ça nous paraît curieux comme procédé, mais c'est courant avec les API. C'est pourquoi cette API peut être utilisée plus facilement via une fonction personnelle, qui elle retournera soit le chemin trouvé, soit vide :

```
'-------------------------------------------------------------------------------
Function RechercheFichierAPI(ByVal StrRepertoire As String, _
                             ByVal StrFichier As String) As String
'-------------------------------------------------------------------------------
' Recherche un fichier dans le répertoire indiqué et ses sous-répertoires.
' Idem Dir() où : ? pour remplacer un caractère et * pour une suite de caractères.
' Retourne : l'adresse du fichier si trouvé, vide si pas trouvé.
'-------------------------------------------------------------------------------
Dim TempStr As String, Ret As Long
TempStr = String(260, 0)
Ret = SearchTreeForFile(StrRepertoire, StrFichier, TempStr)
If Ret <> 0 Then RechercheFichierAPI = Left$(TempStr, InStr(1, TempStr, Chr$(0)) - 1)
End Function
```

Voilà comment résoudre un problème complexe en quelques lignes de code.

La plus connue des API est certainement **Sleep** qui permet de faire une pause d'une durée exprimée en millisecondes. Par exemple pour une pause d'une demi-seconde, **Sleep 500** remplace les instructions :

```
T = Timer : While T + 0.5 > Timer : Wend
```

Elle se déclare ainsi : Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

Le nom de l'API peut être modifié, comme dans la déclaration suivante :

```
Declare Sub Attente Lib "kernel32" Alias Sleep (ByVal dwMilliseconds As Long)
```

Ainsi l'API **Sleep** sera appelée **Attente** dans le code.

L'utilisation des objets ActiveX est une autre façon d'intégrer des fonctionnalités non disponibles en VBA. Ci-dessous, je reprends une fenêtre de message identique à la fonction **MsgBox**, mais avec la particularité de se refermer automatiquement au bout d'un laps de temps déterminé.
Dans ce cas la fonction retourne -1 (ou **True**).
Inversement, si l'utilisateur fait un choix avant la durée impartie, la fonction retourne une valeur de l'énumération VbMsgBoxResult comme le fait **MsgBox**, et bien sûr la fenêtre est fermée.

J'utilise une fonction personnelle, nommée MsgBoxTimer, pour simplifier l'utilisation de cet objet :

```
'-------------------------------------------------------------------------------
Function MsgBoxTimer(Message As String, Boutons As VbMsgBoxStyle, Titre As String, _
                     DuréeAffichage As Byte) As VbMsgBoxResult
'-------------------------------------------------------------------------------
' Affiche une boite de message comme MsgBox, mais avec une durée d'affichage maximale
' qui peut être déterminée.
'-------------------------------------------------------------------------------
' Message : Message à afficher comme pour la fonction MsgBox.
' Boutons : Les mêmes boutons que pour la fonction MsgBox.
' Titre : Le titre de la boite comme pour la fonction MsgBox.
' DuréeAffichage : Durée en secondes d'affichage de la boite. Sauf si un choix est fait.
' Retourne : -1 si pas de choix, ou une constante VbMsgBoxResult comme pour MsgBox.
'-------------------------------------------------------------------------------
Dim InfoBox As Object
Set InfoBox = CreateObject("WScript.Shell")
MsgBoxTimer = InfoBox.PopUp(Message, DuréeAffichage, Titre, Boutons)
End Function
```

Voici un exemple d'appel de cet objet, qui laisse trois secondes à l'utilisateur pour annuler un traitement. Si l'utilisateur clique sur le bouton « Non » alors la procédure est quittée, sinon elle est exécutée :

```
Sub Test()

If MsgBoxTimer("Vous avez 3 secondes pour annuler le traitement.", vbCritical + vbYesNo, _
               "Confirmez-vous le traitement ?", 3) = vbNo Then Exit Sub

Debug.Print "Traitement Confirmé."

End Sub
```

XXIII - QuickRanking - Algorithme rapide de tri et de classement des données

Nous arrivons bientôt au terme de ce mémento. Il est temps de vérifier si nous sommes devenus des champions du VBA en développant un algorithme de tri capable de rivaliser avec le célèbre QuickSort...

L'algorithme QuickSort est le roi du tri rapide depuis un demi-siècle. Par contre il possède trois failles qui peuvent être exploitées pour dresser un cahier des charges de notre futur algorithme :

- QuickSort est un algorithme dit « en place », c'est-à-dire que les données passées sous forme de tableau dans l'argument de la fonction, sont triées directement dans le tableau, sans utiliser de mémoire de stockage annexe, ce qui peut être un avantage. Inversement, il n'est pas possible de savoir quel était l'indice d'origine des données

triées, en d'autres termes, pour obtenir un ordre de classement et non un simple tri, il faut lancer un deuxième traitement, qui augmente d'environ 30 % le temps du traitement global. Mon algorithme devra donc être capable de restituer le tri des données, mais aussi le classement des données, sans traitement supplémentaire ;

- QuickSort est très performant sur des données réparties de façon idéalement aléatoire. Sur les données déjà partiellement triées, l'algorithme devient moins impressionnant. Notre algorithme devra donc être capable de trier rapidement les données déjà partiellement classées ;
- QuickSort est un algorithme dit « non stable », car les données à trier sont déplacées. Donc, plus elles sont volumineuses en taille d'octets, plus le traitement prend du temps. Ainsi trier des chaînes de 250 caractères sera beaucoup plus long que trier des entiers sur 4 octets. Notre algorithme devra donc être capable de trier les données sans les déplacer, ce qui semble contradictoire… mais c'est possible.

Trions les 12 données du tableau ci-dessous pour comprendre cette nouvelle méthode de tri, qui pour économiser du temps de traitement, ne déplace pas les données, mais mémorise pour chaque donnée où est la donnée qui la suit :

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	**3**	**9**	**5**	**2**	**15**	**7**	**8**	**20**	**17**	**0**	**5**	**90**

Analysons les deux premiers éléments : ici l'on sait que l'indice 0, qui vaut 3 est la valeur minimale et l'indice 1 qui vaut 9 est la valeur maximale.

Donc l'indice suivant l'indice 0 est l'indice 1. Nous pouvons mémoriser ces informations :

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	3	9	**5**	**2**	**15**	**7**	**8**	**20**	**17**	**0**	**5**	**90**
Indice suivant :	1	x										

Pour simplifier la lecture du tableau, la valeur minimale est représentée en vert et la valeur maximale est en orange.

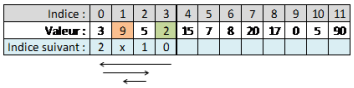
Pour classer l'indice 2, qui vaut 5 : nous lisons la valeur minimale, l'indice 0. La valeur 3 est inférieure à 5, donc nous lisons l'indice suivant, l'indice 1, qui vaut 9. Trop grand cette fois. Donc 5 devient la valeur qui suivra 3 et 9 sera la valeur qui suit 5.

Ce qui met à jour notre tableau des « indices suivants » :

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	3	9	**5**	**2**	**15**	**7**	**8**	**20**	**17**	**0**	**5**	**90**
Indice suivant :	2	x	1									

Classons l'indice 3 qui vaut 2, valeur inférieure à notre ancienne valeur minimale qui était l'indice 0 de valeur 3. Donc un échange est effectué. Et l'indice 0 sera la valeur qui suivra l'indice 3.

Pas de changement pour les autres données déjà analysées :



Ces flèches indiquent l'ordre de lecture du tableau en partant de la référence minimale, et en lisant les « indices suivants ».

Passons à l'indice 4 qui vaut 15, valeur supérieure à notre ancienne valeur maximale, l'indice 1 qui vaut 9. Donc un échange est effectué. L'indice 4 sera la valeur qui suivra l'indice 1.

Pas de changement pour les autres données déjà analysées :

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	**3**	**9**	**5**	**2**	15	**7**	**8**	**20**	**17**	**0**	**5**	**90**
Indice suivant :	2	4	1	0	x							

L'indice 5 vaut 7, valeur supérieure à notre valeur minimale de référence, l'indice 3, qui vaut 2. Donc nous lisons l'indice suivant, l'indice 0, qui vaut 3. Toujours inférieur à 7, donc nous lisons l'indice suivant, l'indice 2, qui vaut 5. Nous continuons avec l'indice suivant, 1, qui vaut 9. Cette fois nous pouvons sortir de la boucle et mettre à jour les indices : l'indice suivant le chiffre 7 est donc l'indice 1, et le nouvel indice suivant de l'indice 2 n'est plus 1 mais 5 :

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	**3**	**9**	**5**	**2**	15	**7**	**8**	**20**	**17**	**0**	**5**	**90**
Indice suivant :	2	4	5	0	x	1						

Mêmes principes pour classer l'indice 6 qui vaut 8 : après lecture des valeurs du tableau en partant de l'indice de la valeur minimale et en utilisant les « indices suivants », nous trouvons qu'il se situe entre l'indice 5 qui vaut 7 et l'indice 1 qui vaut 9. Donc son « indice suivant » est l'indice 1, et « l'indice suivant » de l'indice 5 est modifié : 1 est remplacé par 6.

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Données :	**3**	**9**	**5**	**2**	15	**7**	**8**	**20**	**17**	**0**	**5**	**90**
Indice suivant :	2	4	5	0	x	6	1					

Poursuivez le classement et si vous obtenez ce tableau, c'est que vous avez compris :

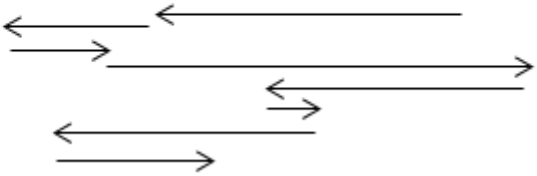
Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	**3**	**9**	**5**	**2**	**15**	**7**	**8**	**20**	**17**	0	**5**	90
Indice suivant :	2	4	10	0	8	6	1	11	7	3	5	x

Pour lire les données dans l'ordre croissant, il faut partir de la référence minimale, l'indice 9, soit la valeur 0, et lire les « indices suivants » : l'indice 3, qui vaut 2, l'indice 0 qui vaut 3, l'indice 2 qui vaut 5, et ainsi de suite…

Les données sont triées sans qu'aucun déplacement n'ait été fait.

Seul problème : pour connaître le classement d'un élément, il faut parfois faire de nombreuses lectures.
Par exemple, pour classer l'indice 12 qui vaut 10, voici le chemin suivi, soit 8 lectures, pour trouver que l'indice 1 est le dernier élément inférieur :

Indice :	0	1	2	3	4	5	6	7	8	9	10	11	12
Valeur :	**3**	**9**	**5**	**2**	**15**	**7**	**8**	**20**	**17**	0	**5**	90	**10**
Indice suivant :	2	4	10	0	8	6	1	11	7	3	5	x	



Vous devinez qu'avec un tableau de plusieurs centaines d'éléments, certaines valeurs à classer vont nécessiter un nombre de lectures impressionnant… ce qui fait exploser les temps de traitement.

Pour gagner du temps, il faut tenir à jour un tableau des données déjà classées :

Indice :	9	3	0	2	10	5	6	1	4	8	7	11
Valeur :	**0**	**2**	**3**	**5**	**5**	**7**	**8**	**9**	**15**	**17**	**20**	**90**

Ce qui permet de faire une recherche dichotomique pour trouver la valeur la plus proche de l'élément que l'on souhaite classer, qui vaut 10, en seulement 4 lectures :

Indice :	9	3	0	2	10	5	6	1	4	8	7	11
Valeur :	**0**	**2**	**3**	**5**	**5**	**7**	**8**	**9**	**15**	**17**	**20**	**90**

Indice :						5	6	1	4	8	7	11
Valeur :						**7**	**8**	**9**	**15**	**17**	**20**	**90**

Indice :						5	6	1				
Valeur :						**7**	**8**	**9**				

Indice :							6	1				
Valeur :							**8**	**9**				

Cette méthode prend toute sa puissance sur un tableau très grand. Par exemple, sur un tableau de 10 000 éléments, 14 lectures seulement permettent de trouver l'élément désiré, soit environ Log(n) / Log(2), où n est le nombre d'éléments du tableau. À comparer avec les centaines de lectures nécessaires par la lecture un à un des « indices suivants ».

La génération de ce tableau des données déjà classées étant chronophage, elle ne sera réalisée que de temps en temps. Mais même incomplet, ce tableau permet de se rapprocher rapidement de l'élément le plus proche, pour ensuite faire un traitement ordinaire en lisant les « indices suivants » un à un, jusqu'à trouver le bon élément.
Considérez ce tableau comme un raccourci pour arriver rapidement proche de la destination finale, et pas forcément à la destination finale.

L'algorithme de recherche dichotomique de QuickRanking se distingue des algorithmes classiques basés sur une boucle de type « **Do… Loop While** *Début* ≤ *Fin* », incluant une condition d'égalité pour sortir de la boucle prématurément lorsque la valeur recherchée est trouvée. Pour accélérer les traitements QuickRanking procède en deux étapes : la première étape est une boucle « **For… Next** » faite un nombre de fois inférieur au nombre calculé par la formule Log(n) / Log(2) qui donne le nombre maximum de découpages qu'il faudra faire pour trouver la solution dans une recherche dichotomique. Ce qui est suffisant pour s'approcher de la solution ; la seconde étape est une boucle qui part de la dernière solution inférieure et qui, cette fois, recherche la solution exacte.

Cette approche permet de limiter le nombre de tests conditionnels, qui sont gourmands en temps de traitement.

Reprenons le tableau déjà vu :

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	**3**	**9**	**5**	**2**	**15**	**7**	**8**	**20**	**17**	0	**5**	90
Indice suivant :	2	4	10	0	8	6	1	11	7	3	5	x

La lecture des indices donne le tri des éléments : 0, 2, 3, 5, 5, 7, 8, 9, 15, 17, 20, 90.

Cette lecture permet aussi d'obtenir l'ordre de classement des éléments :

Le premier élément est l'indice 9. Dans un tableau annexe qui sert d'ordre de classement, l'on met 1 pour l'indice 9.

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	**3**	**9**	**5**	**2**	**15**	**7**	**8**	**20**	**17**	0	5	90
Indice suivant :	2	4	10	0	8	6	1	11	7	3	5	x
Classement :										1		

L'indice suivant est l'indice 3, l'on met 2 dans l'ordre de classement de l'indice 3.

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	**3**	**9**	**5**	**2**	**15**	**7**	**8**	**20**	**17**	0	5	90
Indice suivant :	2	4	10	0	8	6	1	11	7	3	5	x
Classement :				2						1		

L'indice suivant est l'indice 0, l'on met 3 dans l'ordre de classement de l'indice 0.

Et ainsi de suite pour obtenir ce tableau :

Indice :	0	1	2	3	4	5	6	7	8	9	10	11
Valeur :	**3**	**9**	**5**	**2**	**15**	**7**	**8**	**20**	**17**	0	5	90
Indice suivant :	2	4	10	0	8	6	1	11	7	3	5	x
Classement :	3	8	4	2	9	6	7	11	10	1	5	12

L'indice 0, qui vaut 3, est le 3^{e} élément de la liste classée.

L'indice 1 qui vaut 9 est le 8^{e} élément de la liste classée.

L'ordre de classement est pratique pour remplir un tableau de ce type :

Participant	Epreuve 1	Classement	Epreuve 2	Classement	Total	Classement
Mathieu	**15,24**	3	**12,65**	4	**27,89**	3
Sylvie	**14,15**	4	**17,45**	1	**31,60**	2
Nathalie	**16,24**	2	**8,59**	5	**24,83**	5
Edouard	**17,50**	1	**14,15**	2	**31,65**	1
Luc	**13,67**	5	**13,80**	3	**27,47**	4

Mais l'exercice peut se révéler plus compliqué dans certains cas, comme pour le tableau ci-dessous :

Indice :	0	1	2	3	4	5	6	7	8
Valeur :	**8**	**8**	**9**	**8**	**10**	**8**	**8**	10	5
Indice suivant :	1	2	4	0	7	3	5	x	6
Classement :	5	6	7	4	8	3	2	9	1

Les éléments de valeurs 8, des indices 0, 1, 3, 5 et 6, ont un ordre de classement incohérent. Rien ne peut justifier que l'indice 6 soit classé 2^{e} alors que l'indice 5 est classé 3^{e}.

Dans la pratique, soit l'on attribue le même classement à toutes les valeurs égales :

Indice :	0	1	2	3	4	5	6	7	8
Valeur :	**8**	**8**	**9**	**8**	**10**	**8**	**8**	10	5
Indice suivant :	1	2	4	0	7	3	5	x	6
Classement :	2	2	7	2	8	2	2	8	1

Ici il y a un premier, et 5 deuxièmes ex aequo. L'élément de valeur 9 (indice 2) est 7^{e}.

Soit l'on conserve l'ordre d'origine pour classer les valeurs égales :

Indice :	0	1	2	3	4	5	6	7	8
Valeur :	**8**	**8**	**9**	**8**	**10**	**8**	**8**	10	5
Indice suivant :	1	2	4	0	7	3	5	x	6
Classement :	2	3	7	4	8	5	6	9	1

Ici chaque élément à un rang distinct. Il n'y a pas d'ex aequo même pour les éléments de même valeur. En cas d'égalité, c'est l'élément à l'indice le plus bas qui est priorisé.

La fonction accepte quatre arguments, en plus du tableau à trier :

- OrdreCroissant : s'il vaut **True**, le tri se fait par ordre croissant, s'il vaut **False**, le tri est décroissant ;
- ModeClassement indique le mode de classement à retourner : 0 pour aucun classement, 1 pour appliquer le même rang aux données égales, 2 pour attribuer un rang différent aux données égales en respectant leur position d'origine, 3 pour retourner uniquement le classement (sans trier), 4 pour trier en supprimant les doublons (mais sans retourner le classement) ;
- NiveauTest : s'il vaut **False**, aucune analyse supplémentaire n'est réalisée. S'il vaut **True** des tests supplémentaires sont réalisés en se basant sur le dernier élément analysé : recherche des suites et des doublons pour un traitement immédiat. S'il vaut entre 1 et 100, lance le test pour savoir s'il faut activer ou non l'option, où NiveauTest représente le taux de conformité souhaitée (de 1 à 100) pour que l'activation de l'option soit considérée utile ;
- TauxTest : contiendra le taux (0 à 100) des tests efficaces. Utilisé pour tester l'option.

EXCEL propose deux fonctions intégrées pour retourner le classement des données :

« **=EQUATION.RANG(Nombre,Référence,[Ordre])** » anciennement « **=RANG(Nombre,Référence,[Ordre])** » :

- Nombre : est la valeur numérique à analyser ;
- Référence : est la référence du tableau contenant l'ensemble des valeurs ;
- Ordre : est l'ordre de classement, 0 = décroissant, 1 = croissant.

Cette fonction attribue le même classement à toutes les valeurs égales, et attention, elle ne gère pas les valeurs alphabétiques. Pour obtenir le classement des valeurs alphabétiques, vous devez dans un premier temps, les copier dans un tableau annexe et trier ce tableau. Dans un second temps, utiliser la fonction « **=EQUIV(Valeur,Référence,0)** ». Cette fonction attribue le même classement à toutes les valeurs égales. Elle gère aussi les valeurs numériques.

Cependant, ces fonctions ont des temps de traitement énormes et sont difficilement exploitables sur des tableaux de plus de 50 000 lignes.

Le code de la procédure QuickRanking :

```
'-------------------------------------------------------------------------------
Public Function QuickRanking(ByRef TabDonnées() As Variant, _
                Optional ByVal OrdreCroissant As Boolean = True, _
                Optional ByVal ModeClassement As Byte = 1, _
                Optional ByRef NiveauTest As Long = 15, _
                Optional ByRef TauxTest As Long = 0) As Variant
'-------------------------------------------------------------------------------
' TabDonnées : Trie les données passées en argument et modifie TabDonnées.
' OrdreCroissant : Si vaut True alors ordre croissant, sinon ordre décroissant.
' ModeClassement : 0 = Tri, pas de classement.
'                  1 = Tri + classement des données, les données égales ont le même ordre.
'                  2 = Tri + classement des données, l'ordre des données égales
'                      respecte l'ordre d'origine.
'                  3 = Uniquement classement des données, et sans gestion des égalités.
'                  4 = Tri sans doublon, et sans classement.
' NiveauTest : False (0) = Pas de test complémentaire,
'              True (-1) = Contrôle les égalités et les suites.
'              >0 et <100 = Lance le test pour savoir s'il faut activer ou non l'option,
'                           où NiveauTest représente le taux de conformité (de 1 à 100)
'                           pour que l'activation de l'option soit considérée utile.
'                           NiveauTest sera alimenté du résultat obtenu (Vrai ou Faux).
' TauxTest : Contiendra le taux (0 à 100) des tests efficaces. Utilisé pour tester l'option.
```

```
'-------------------------------------------------------------------------------

' S'il faut lancer le test du choix de l'option pour NiveauTest. NiveauTest contient
' le pourcentage de réussite désiré des tests complémentaires pour activer l'option :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If NiveauTest > 0 Then NiveauTest = TesterNiveauQR(TabDonnées(), NiveauTest)

' Bornes du tableau des données d'origine :
Dim TabDébut As Long, TabFin As Long
On Error Resume Next ' Si aucune donnée à trier.
TabDébut = LBound(TabDonnées)
TabFin = UBound(TabDonnées)

' Initialisation du tableau du classement des données :
ReDim Ref(TabDébut - 2 To TabFin) As Long

' Si rien à trier alors quitte :
If Abs(TabFin - TabDébut) < 1 Then QuickRanking = Ref(): Exit Function

' Initialisation des variables pour le traitement de tri :
Dim Tps As Variant, ValMini As Variant, ValMaxi As Variant
Dim i As Long, n As Long, j As Long, Anc As Long, l As Long
Dim RefMini As Long, RefMaxi As Long, MaxiRac As Long, MiniRac As Long
Dim NbPassage As Long, Début As Long, Fin As Long
Dim NbRechercheDicho As Long, MaxiDoWhile As Long, Compteur As Long

' Initialisation du tableau des données déjà classées :
ReDim TabTps(TabDébut - 2 To TabFin) As Long
MaxiRac = TabDébut
NbPassage = TabFin

' Configure le classement des 2 premiers éléments :
If TabDonnées(TabDébut) > TabDonnées(TabDébut + 1) Then n = 1
RefMini = TabDébut + n
RefMaxi = TabDébut + 1 - n
Ref(TabDébut) = TabDébut + 1
Ref(TabDébut + 1) = RefMaxi
ValMini = TabDonnées(RefMini)
ValMaxi = TabDonnées(RefMaxi)

' Si l'option des tests complémentaires est à Vrai (-1) :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If NiveauTest = True Then

' Boucle sur les éléments à classer en effectuant les tests complémentaires :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
For n = 2 + TabDébut To TabFin
    Tps = TabDonnées(n)

   ' Controle le débordement du mini :
    Do Until Tps > ValMini            ' Plus rapide que If Tps <= ValMini Then... End If.
        Ref(n) = RefMini              ' La donnée suivante de n est l'ancien minimum.
        RefMini = n                   ' Mémorise qui est le nouveau minimum.
        TabTps(TabDébut) = n          ' Le 1er élément du tableau de recherche dicho.
        MiniRac = TabDébut            ' Le minimum pour la mise à jour du tableau de recherche
 dicho.
        Anc = TabDébut                ' Position du dernier élément analysé dans le tableau de
 recherche dicho.
        ValMini = Tps                 ' Nouveau minimum.
        GoTo Element_Suivant          ' Fin du traitement de n.
    Loop

    ' Controle le débordement du maxi :
    Do Until ValMaxi > Tps           ' Plus rapide que If Tps >= ValMaxi Then... End If.
        Ref(RefMaxi) = n             ' La donnée suivante de l'ancien maximum est n.
        Ref(n) = n                   ' La donnée suivante de n est n.
        RefMaxi = n                  ' Mémorise qui est le nouveau maximum.
        MaxiRac = MaxiRac + 1        ' Dernière positon dans le tableau de recherche dicho.
        TabTps(MaxiRac) = n          ' Le tableau de recherche dicho peut être alimenté.
        Anc = MaxiRac                ' Position du dernier élément analysé dans le tableau de
 recherche dicho.
        ValMaxi = Tps                ' Nouveau maximum.
```

```
        GoTo Element_Suivant        ' Fin du traitement de n.
    Loop

    ' Mise à jour du tableau des données déjà classées :
    While NbPassage > n                         ' While est plus rapide que If... Then... End If.
        i = TabTps(MiniRac)                     ' Boucle depuis la position du plus petit élément
 analysé,
        If MiniRac = TabDébut Then i = RefMini ' ou boucle depuis la position du minimum.
        For j = MiniRac To n
            TabTps(j) = i                       ' Mémorise la position de l'élément.
            i = Ref(i)                          ' Position de l'élément suivant.
        Next j
        MaxiRac = n - 1                         ' Le dernier élément n'est pas utilisé.
        MiniRac = MaxiRac                       ' Efface la position du plus petit élément
 analysé.
        NbPassage = n * 0.3                     ' Initialise le nombre de passages pour mise à
 jour du tableau.
        NbRechercheDicho = Log(n) / Log(2)      ' Nombre maximum de recherches possibles dans le
 tableau dicho.
        If NbRechercheDicho > 5 Then MaxiDoWhile = NbRechercheDicho ' Limite pour les suites non
 contigües
        Début = TabDébut: Fin = MaxiRac         ' Bornes pour la recherche dichotomique.
        GoTo RechercheDichotomique
    Wend

    ' Bornes pour la recherche dichotomique dans le tableau des données déjà classées :
    Début = TabDébut: Fin = MaxiRac

    ' Tests complémentaires (égalités et suites immédiates) :
    Do Until TabDonnées(n - 1) > Tps            ' Si n est >= dernier élément analysé.
        Début = Anc                             ' Borne de début pour la recherche dicho.
        Do Until Tps > TabDonnées(Ref(n - 1))   ' Si n est <= élément suivant du dernier élément
 analysé.
            Ref(n) = Ref(n - 1)                 ' Échange de la donnée suivante de n et de
 l'ancien élément.
            Ref(n - 1) = n                      ' n devient la donnée suivante de l'ancien
 élément.
            TauxTest = TauxTest + 1             ' Nombre de tests efficaces.
            GoTo Element_Suivant                ' Fin du traitement de n.
        Loop
        GoTo RechercheDichotomique              ' Passe à la recherche avec la nouvelle borne de
 début.
    Loop
    Fin = Anc                                   ' Borne de fin pour la recherche dicho.

    ' Recherche dichotomique dans le tableau des données déjà classées :
RechercheDichotomique:

    For j = 4 To NbRechercheDicho  ' Plus rapide que Do...Loop While Début + 2 < Fin
        i = (Début + Fin) / 2      ' Calcule le milieu.
        If Tps > TabDonnées(TabTps(i)) Then Début = i Else Fin = i
    Next j
    While TabDonnées(TabTps(Début + 1)) < Tps: Début = Début + 1: Wend

    Anc = Début     ' Solution.
    i = TabTps(Anc) ' Plus proche donnée inférieure connue.
    While Anc < MiniRac: MiniRac = Anc: Wend ' Plus rapide que If Anc < MiniRac Then MiniRac =
 Anc

    ' Boucle sur les indices suivants pour trouver le classement du nouvel élément :
    Compteur = 0
    Do
        j = i                       ' Dernière solution.
        i = Ref(i)                  ' Indice suivant
        Compteur = Compteur + 1     ' Compte le nombre de passages infructueux.
    Loop While Tps > TabDonnées(i)  ' Sort si la valeur de l'indice suivant >= Tps.
    NbPassage = NbPassage + Compteur

    Ref(n) = Ref(j)        ' Qui est la donnée suivante de n.
    Ref(j) = n             ' n devient la donnée suivante de l'ancien élément.

    ' Gestion des suites non contigües :
```

```
    While Compteur > MaxiDoWhile
        TabTps(Anc - 2) = TabTps(Anc - 1)
        TabTps(Anc - 1) = TabTps(Anc)
        TabTps(Anc) = n
        TabTps(TabDébut) = RefMini
        Compteur = MaxiDoWhile
    Wend

Element_Suivant:
Next n

' Alimente le taux d'efficacité des tests complémentaires :
TauxTest = TauxTest * 100 / (TabFin - TabDébut)

' Si l'option des tests complémentaires est à Faux (0) :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Else

' Boucle sur les éléments à classer sans effectuer les tests complémentaires :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
For n = 2 + TabDébut To TabFin
    Tps = TabDonnées(n)

   ' Controle le débordement du mini :
    Do Until Tps > ValMini           ' Plus rapide que If Tps <= ValMini Then... End If.
        Ref(n) = RefMini             ' La donnée suivante de n est l'ancien minimum.
        RefMini = n                  ' Mémorise qui est le nouveau minimum.
        TabTps(TabDébut) = n         ' Le 1er élément du tableau de recherche dicho.
        MiniRac = TabDébut           ' Le minimum pour la mise à jour du tableau de recherche
 dicho.
        Anc = TabDébut               ' Position du dernier élément analysé dans le tableau de
 recherche dicho.
        ValMini = Tps                ' Nouveau minimum.
        GoTo ST_Element_Suivant        ' Fin du traitement de n.
    Loop

    ' Controle le débordement du maxi :
    Do Until ValMaxi > Tps          ' Plus rapide que If Tps >= ValMaxi Then... End If.
        Ref(RefMaxi) = n            ' La donnée suivante de l'ancien maximum est n.
        Ref(n) = n                  ' La donnée suivante de n est n.
        RefMaxi = n                 ' Mémorise qui est le nouveau maximum.
        MaxiRac = MaxiRac + 1       ' Dernière positon dans le tableau de recherche dicho.
        TabTps(MaxiRac) = n         ' Le tableau de recherche dicho peut être alimenté.
        Anc = MaxiRac               ' Position du dernier élément analysé dans le tableau de
 recherche dicho.
        ValMaxi = Tps               ' Nouveau maximum.
        GoTo ST_Element_Suivant       ' Fin du traitement de n.
    Loop

    ' Mise à jour du tableau des données déjà classées :
    While NbPassage > n                       ' While est plus rapide que If... Then... End If.
        i = TabTps(MiniRac)                   ' Boucle depuis la position du plus petit élément
 analysé,
        If MiniRac = TabDébut Then i = RefMini ' ou boucle depuis la position du minimum.
        For j = MiniRac To n
            TabTps(j) = i                     ' Mémorise la position de l'élément.
            i = Ref(i)                        ' Position de l'élément suivant.
        Next j
        MaxiRac = n - 1                       ' Le dernier élément n'est pas utilisé.
        MiniRac = MaxiRac                     ' Efface la position du plus petit élément
 analysé.
        NbPassage = 0                         ' Initialise le nombre de passages pour mise à
 jour du tableau.
        NbRechercheDicho = Log(n) / Log(2)    ' Nombre maximum de recherches possibles dans le
 tableau dicho.
    Wend

    ' Recherche dichotomique dans le tableau des données déjà classées :
    Début = TabDébut: Fin = MaxiRac
    For j = 2 To NbRechercheDicho ' Plus rapide que Do...Loop While Début + 2 < Fin
        i = (Début + Fin) / 2      ' Calcule le milieu.
        If Tps > TabDonnées(TabTps(i)) Then Début = i Else Fin = i
```

```
    Next j

    Anc = Début     ' Solution.
    i = TabTps(Anc) ' Plus proche donnée inférieure connue.
    While Anc < MiniRac: MiniRac = Anc: Wend ' Plus rapide que If Anc < MiniRac Then MiniRac =
 Anc

    ' Boucle sur les indices suivants pour trouver le classement du nouvel élément :
    Do
        j = i                       ' Dernière solution.
        i = Ref(i)                  ' Indice suivant
        NbPassage = NbPassage + 1   ' Compte le nombre de passages infructueux.
    Loop While Tps > TabDonnées(i)  ' Sort si la valeur de l'indice suivant >= Tps.

    Ref(n) = Ref(j)       ' Qui est la donnée suivante de n.
    Ref(j) = n            ' n devient la donnée suivante de l'ancien élément.

ST_Element_Suivant:
Next n

End If

' S'il faut retourner le Classement sans le tri :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If ModeClassement = 3 Then
    i = TabTps(MiniRac): If MiniRac = TabDébut Then i = RefMini
    For n = MiniRac To TabFin
        TabTps(n) = i
        i = Ref(i)
    Next n
    QuickRanking = TabTps()
    Exit Function
End If

' Fait une copie temporaire du tableau d'origine :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Erase TabTps: ReDim Mémo(TabDébut To TabFin) As Variant
For n = TabDébut To TabFin
    Mémo(n) = TabDonnées(n)
Next n

' Initialisation du tableau du classement si demandé :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If ModeClassement > 0 Then
    ReDim Pos(TabDébut To TabFin) As Long
    ReDim Egalités(TabDébut To TabFin) As Long
End If

' Classe les données dans l'ordre croissant :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If OrdreCroissant = True Then
    i = RefMini
    For n = TabDébut To TabFin
        TabDonnées(n) = Mémo(i)
        i = Ref(i)
    Next n

    ' S'il faut retourner le Classement où les égalités ont le même classement :
    If ModeClassement = 1 Then
        i = RefMini: Anc = i: NbPassage = 1
        For n = TabDébut To TabFin
            Pos(i) = NbPassage: NbPassage = NbPassage + 1
            If Mémo(i) = Mémo(Anc) Then Pos(i) = Pos(Anc)
            Anc = i: i = Ref(i)
        Next n
        QuickRanking = Pos(): Exit Function
    End If

    ' S'il faut retourner le classement où les égalités distinguent l'ordre d'origine :
    If ModeClassement = 2 Then
        i = RefMini: Anc = i: j = TabDébut: NbPassage = 1
        For n = TabDébut To TabFin
```

```
            Egalités(j) = i: Anc = i: i = Ref(i): j = j + 1
            If Mémo(i) > Mémo(Anc) Then
                If j > TabDébut + 1 Then Call QuickSort(Egalités(), TabDébut, j - 1)
                For l = TabDébut To j - 1
                    Pos(Egalités(l)) = NbPassage: NbPassage = NbPassage + 1
                Next l
                j = TabDébut
            End If
        Next n
        If j > TabDébut + 1 Then Call QuickSort(Egalités(), TabDébut, j - 1)
        For l = TabDébut To j - 1
            Pos(Egalités(l)) = NbPassage: NbPassage = NbPassage + 1
        Next l
        QuickRanking = Pos(): Exit Function
    End If

    ' S'il faut retourner le tri sans doublon (et sans classement) :
    If ModeClassement = 4 Then
    NbPassage = TabDébut
    For n = TabDébut + 1 To TabFin
        If TabDonnées(n) <> TabDonnées(n - 1) Then NbPassage = NbPassage + 1
        TabDonnées(NbPassage) = TabDonnées(n)
    Next n
    ReDim Preserve TabDonnées(TabDébut To NbPassage)
        QuickRanking = Pos(): Exit Function
    End If

    QuickRanking = Pos()
    Exit Function

End If

' Classe les données dans l'ordre décroissant :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
i = RefMini
For n = TabFin To TabDébut Step -1
    TabDonnées(n) = Mémo(i)
    i = Ref(i)
Next n

' S'il faut retourner le classement où les égalités ont le même classement :
If ModeClassement = 1 Then
    i = RefMini: Anc = i: NbPassage = TabFin - TabDébut + 1
    For n = TabFin To TabDébut Step -1
        Pos(i) = NbPassage: NbPassage = NbPassage - 1
        If Mémo(i) = Mémo(Anc) Then Pos(i) = Pos(Anc)
        Anc = i
        i = Ref(i)
    Next n
    QuickRanking = Pos(): Exit Function
End If

' S'il faut retourner le classement où les égalités distinguent l'ordre d'origine :
If ModeClassement = 2 Then
    i = RefMini: Anc = i: j = TabDébut: NbPassage = TabFin - TabDébut + 1
    For n = TabDébut To TabFin
        Egalités(j) = i
        Anc = i
        i = Ref(i)
        j = j + 1
        If Mémo(i) <> Mémo(Anc) Or n = TabFin Then
            If j > TabDébut + 1 Then Call QuickSort(Egalités(), TabDébut, j - 1)
            For l = TabDébut To j - 1
                Pos(Egalités(l)) = NbPassage
                NbPassage = NbPassage - 1
            Next l
            j = TabDébut
        End If
    Next n
    QuickRanking = Pos(): Exit Function
End If
```

```
' S'il faut retourner le tri sans doublon (et sans classement) :
If ModeClassement = 4 Then
    NbPassage = TabDébut
    For n = TabDébut + 1 To TabFin
        If TabDonnées(n) <> TabDonnées(n - 1) Then NbPassage = NbPassage + 1
        TabDonnées(NbPassage) = TabDonnées(n)
    Next n
    ReDim Preserve TabDonnées(TabDébut To NbPassage)
    QuickRanking = Pos(): Exit Function
End If

QuickRanking = Pos()
End Function
```

QuickRanking propose en option des analyses complémentaires pour accélérer les traitements, en se basant sur le dernier élément analysé - recherche des suites et des doublons pour un traitement immédiat.
Ces analyses sont performantes sur les listes partiellement classées ou s'il y a de nombreuses égalités, mais sont contre-productives sur les listes aléatoires.
C'est pourquoi ces analyses sont optionnelles dans QuickRanking afin de ne pas les exécuter sur les listes de données que vous savez aléatoires, et ainsi économiser du temps de traitement.

Le dilemme se pose donc lorsque vous ne savez pas à l'avance si la liste des données à analyser sera ou non aléatoire. Car dans certains cas, l'activation de l'option accélérera incroyablement les traitements, ou au contraire les ralentira.
La solution que je propose, est de prendre un échantillon des données de la liste, pris au hasard, et de l'analyser avec l'option activée. QuickRanking alimentera, dans son argument passé en référence, TauxTest, le taux de tests efficaces. Si ce nombre représente un pourcentage significatif, passé dans l'argument TauxConformité de la fonction, l'on peut estimer qu'il faut activer l'option.

Plus la taille de l'échantillon est grande et plus le test sera représentatif, mais plus il prendra de temps. Ainsi, j'ai limité la taille de l'échantillon à 3 % de la taille de la liste d'origine. Inversement si l'échantillon fait moins de 20 éléments, il est jugé non significatif et la fonction retourne True sans faire de test. Par défaut la taille de l'échantillon est 1/1000 de la liste.

```
'-------------------------------------------------------------------------------
Private Function TesterNiveauQR(ByRef MonTableau() As Variant, _
                        ByVal TauxConformité As Long, _
                        Optional ByVal PcEchantillon As Double = 0.1) As Boolean
'-------------------------------------------------------------------------------
Dim Début As Long, Fin As Long, TailleEchantillon As Long, i As Long, l As Long
Début = LBound(MonTableau())
Fin = UBound(MonTableau())

' Initialisation des variables :
TailleEchantillon = (Fin - Début) * PcEchantillon / 100

' Contrôle la taille de l'échantillon pris au hasard dans la liste :
If TailleEchantillon > Fin * 0.03 Then TailleEchantillon = Fin * 0.03
If TailleEchantillon < 20 Then TesterNiveauQR = True: Exit Function
ReDim MonTest(Début To TailleEchantillon) As Variant
Do
    i = Rnd() * Fin
Loop While Début + i + TailleEchantillon > Fin

For l = i To i + TailleEchantillon
    MonTest(Début) = MonTableau(l): Début = Début + 1
Next l

' Compte le nombre de tests fructueux avec l'option à Vrai :
i = 0: Call QuickRanking(MonTest(), True, 3, True, i)

' Retourne Vrai si les tests sont efficaces dans au moins TauxConformité% des cas :
If i > TauxConformité Then TesterNiveauQR = True

End Function
```

Pour remédier au principal défaut de l'algorithme QuickSort lors du tri des données alphabétiques, j'ai écrit une version « enrichie » de QuickSort que j'ai appelée QuickSort_AndRank, qui ne déplace plus directement les données, mais à l'instar de QuickRanking, déplace les références des indices des données. D'où un gain de temps, car l'on travaille alors sur des entiers de quatre octets et non plus sur des chaînes de caractères de taille variable. Cela nécessite évidemment des capacités mémoire supplémentaires, mais permet du coup d'utiliser ces tableaux de référence pour retourner un ordre de classement en plus du tri, tout comme le fait QuickRanking.

J'ai comparé ces trois algorithmes, QuickSort, QuickSort_AndRank, et QuickRanking, sur des listes de données aléatoires :

- données numériques : QuickSort est l'algorithme le plus rapide, sauf lorsque les données en début de liste sont déjà classées, ou lorsque la valeur minimale ou maximale est souvent représentée, ou lorsque l'option des analyses complémentaires peut être utilisée sur les listes partiellement classées ou contenant des doublons. Dans ces cas, QuickRanking prend l'avantage ;

- données alphanumériques : QuickRanking est plus rapide que QuickSort et que QuickSort_AndRank. Avantage accentué lorsque l'option des analyses complémentaires peut être utilisée sur les listes partiellement classées ou contenant des doublons. Inversement, l'avantage s'émousse sur les listes très volumineuses, ou pour le traitement des petites chaînes de caractères.

Conclusion : liste idéalement aléatoire ou liste fortement classée, données volumineuses ou de taille réduite, la réalité se situe souvent entre ces extrêmes. Dans la pratique, QuickRanking offre alors un choix très intéressant…

Découvrez l'algorithme QuickSort_AndRank, consultez la dernière version de QuickRanking et faites vos tests dans le fichier EXCEL en pièce jointe : « **https://hal.archives-ouvertes.fr** » (recherchez avec le mot clé : QuickRanking). Ou consultez le PDF complet : « **https://hal.archives-ouvertes.fr/hal-01154432/document** ».

XXIV - Diffuser un module protégé

Notre module VBO est désormais bien fourni de procédures très pratiques et nous souhaitons le diffuser pour qu'il soit réutilisé dans d'autres applications, mais sans en divulguer le code.
Nous allons créer un fichier « .xlam » qui deviendra une nouvelle référence de la bibliothèque d'objets.

En gardant ouvert le classeur contenant le module « VBO », ouvrir un nouveau classeur EXCEL, et y copier le module « VBO » en le faisant glisser avec la souris.

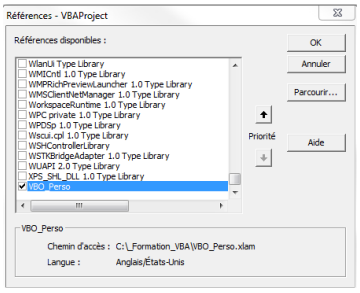
Dans le menu de l'éditeur, choisissez « Outils », puis « Propriétés de VBA_Projet… ».
Changez le nom du projet, par exemple « VBO_Perso ». Dans l'onglet « Protection », cochez « Verrouiller le projet pour l'affichage », donnez un mot de passe qu'il faut confirmer, puis cliquez sur « OK ».
Gardez cependant à l'esprit que les hackers possèdent des outils qui peuvent faire sauter votre protection.

Sauvegardez le classeur au format « .xlam » dans le répertoire de votre application, en lui donnant un nom explicite, par exemple « VBO_Perso.xlam », puis quittez.

Pour importer votre module VBO dans une autre application EXCEL :

depuis l'éditeur, ouvrez le menu « Outils », puis « Références ». Cliquez sur « Parcourir… » ;

dans le type de fichiers, piochez « Tous les fichiers » puis sélectionnez votre fichier « VBO_Perso.xlam » ;



une nouvelle référence est ajoutée à la liste des références disponibles ;

fermez le formulaire en cliquant sur « OK ».

Les procédures de votre module sont désormais disponibles pour cette nouvelle application et le code est protégé.

N'oubliez pas de diffuser ce fichier « VBO_Perso.xlam » avec vos applications EXCEL qui l'utilisent.

Cette pratique permet aussi de simplifier la maintenance de vos applications.
Par exemple, si vous modifiez le code d'une procédure de votre module, il suffit de diffuser le nouveau fichier « VBO_Perso.xlam » aux utilisateurs sans avoir à intervenir sur le code source de leur application.

XXV - Conclusion

De nombreuses notions ont été abordées à travers ces pages, certaines sont simples à comprendre, d'autres vous semblent complexes : rassurez-vous, avec un peu de pratique, en consultant l'aide EXCEL, en analysant le code généré par les macros, vous maîtriserez bientôt les subtilités du VBA.

Vous avez envie d'en savoir plus sur le VBA, ou la programmation en général : faites un tour sur le site « developpez.com », qui est une référence incontournable.
Vous y trouverez ce genre d'astuce pour lancer GoogleMaps d'après une adresse passée en argument :

```
'-------------------------------------------------------------------------------
Public Function CarteGoogleMaps(StrAdresse As String, StrCP As String, _
                                StrVille As String) As Boolean
'-------------------------------------------------------------------------------
' Ouvre Google Maps sur l'adresse passée en argument.
' Exemple d'appel : Call VBO.CarteGoogleMaps("22 rue Cassette", "75006", "Paris")
'-------------------------------------------------------------------------------
Dim v_http As String
v_http = "http://maps.google.fr/maps?f=q&hl=fr&q=" _
         & StrAdresse & " ,+ " & StrCP & "+" & StrVille

On Error GoTo Fin:
ThisWorkbook.FollowHyperlink v_http
CarteGoogleMaps = True
Fin:
End Function
```

Ainsi qu'un tutoriel bien fait : « **ftp://ftp-developpez.com/bidou/Cours/VBA/formationVBA.pdf** ».
Un cours complet pour « Comprendre les Recordset ADO » de Jean-Marc Rabilloud : « **https://vb.developpez.com/bidou/recordset-ado/** ».
Et pour tout savoir sur les imprimantes : « **https://excel.developpez.com/faq/?page=Impression#ProprietesImprimantes** ».

Avant de se quitter, je vous propose en annexe une libre adaptation en VBA du jeu d'arcade « Snake » qui a connu ses heures de gloire au siècle dernier. Pour ceux qui ne le connaissent pas, je vous rappelle le principe : avec les touches du clavier « flèche gauche » et « flèche droite », vous déplacez dans un labyrinthe un serpent qui doit manger dix pommes pour passer au niveau suivant. Pour compliquer la tâche du joueur, chaque pomme mangée rallonge le serpent et les niveaux comportent de plus en plus d'obstacles à éviter.

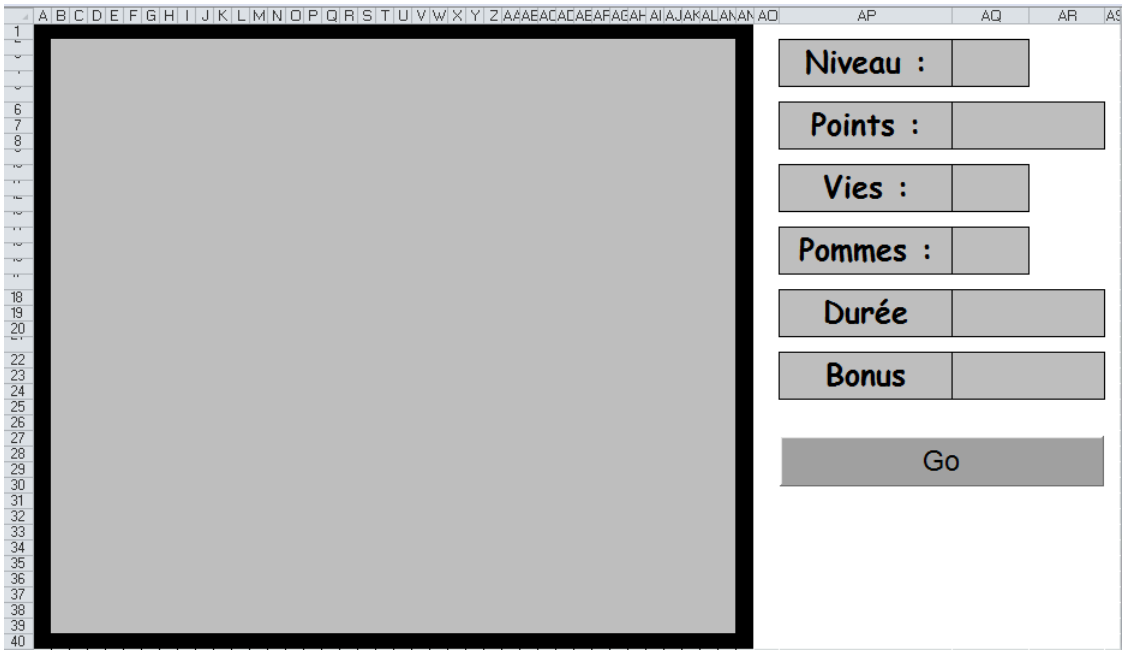
Le code présenté pose les bases de la programmation, et ne demande qu'à être amélioré par votre touche personnelle. Par exemple : ajouter des niveaux, mettre des effets sonores (avec l'API **SndPlaySoundA**, plus connue sous **PlaySound**), mettre des puits pour passer d'un point à l'autre du plateau, faire varier la longueur de la queue ou inverser les touches gauche et droite. Sans oublier la gestion des records, en points et durée, de la partie et des niveaux. Vous avez de quoi vous occuper.

Avant de passer à une version « Graphique », mais c'est une autre histoire, que l'on étudiera dans le prochain mémento.

Laurent OTT. 2016

Annexe : Snake - Programmer un jeu d'arcade en VBA

Le plateau du jeu est constitué d'un carré de 40 sur 40 cellules. Les lignes 1 à 40 ont une hauteur de 12,75 et les colonnes A à AN ont une largeur de 2. Les extrémités du plateau sont renseignées par « X » avec une couleur de fond noire. L'intérieur de plateau est en fond gris. La partie AO1 à AS40 servira à afficher les informations sur le jeu. Les autres lignes et colonnes sont masquées :



La grande difficulté dans la programmation d'un jeu d'arcade est d'obtenir une bonne fluidité de l'animation : si entre chaque affichage du serpent une pause est nécessaire pour éviter de faire avancer le serpent trop vite, cette pause ne doit pas durer trop longtemps pour éviter un effet saccadé. Ici, la pause de 16 centièmes de secondes (qu'il faudra peut-être adapter à la vitesse de votre ordinateur) est mise à profit pour lire les interventions faites au clavier par le joueur.

Pour la lecture du clavier, nous utilisons l'API **GetKeyState**, en passant en argument le numéro de la touche que l'on souhaite tester. La fonction retourne un chiffre négatif si la touche testée est enfoncée. Il faut donc appeler cette fonction pour chacune des touches que l'on souhaite analyser.
J'ai rajouté les touches « Ctrl » pour mettre un turbo, « Suppr » pour faire une téléportation aléatoire, et « Fin » pour finir le jeu prématurément.

Pour minimiser les temps de traitement, nous n'utiliserons que les affichages classiques d'EXCEL, sans incorporer d'effets graphiques.

Pour faciliter la maintenance du programme, le code est réparti dans trois modules.

Le module « Variables » regroupe les déclarations API, les constantes, les variables publiques, et l'initialisation des variables.

```
'-------------------------------------------------------------------------------------------
Declare Function GetKeyState Lib "user32" (ByVal nVirtKey As Long) As Integer
Declare Sub Sleep Lib "Kernel32" (ByVal dwMilliseconds As Long)
```

```
'-------------------------------------------------------------------------------------------------
Public Const EcranMiniY As Integer = 2          ' Y mini du plateau
Public Const EcranMaxiY As Integer = 38         ' Y maxi du plateau
Public Const EcranMiniX As Integer = 2          ' X mini du plateau
Public Const EcranMaxiX As Integer = 38         ' X maxi du plateau
Public Const FormePomme As String * 1 = "@"     ' Forme d'une pomme
Public Const FormePorte As String * 1 = "¤"     ' Forme d'une porte
Public Const NbPommeParNiveau As Byte = 10      ' Nombre de pommes par niveau
Public Const PointParPomme As Byte = 10         ' Points gagnés par pomme mangée
Public Const AvanceParPomme As Byte = 7         ' Allonge du serpent par pomme mangée
'-------------------------------------------------------------------------------------------------
Public MvX As Integer, MvY As Integer   ' Déplacement en X et Y : -1, 0 ou 1
Public Direction As Integer             ' Nouvelle direction: -1 = gauche, +1 = droite
Public i As Integer                     ' Utilisée dans les boucles.
Public TailleSerpent As Byte            ' Taille du corps du serpent
Public Niveau As Byte                   ' Niveau
Public Points As Long                   ' Nombre de points gagnés
Public Vie As Integer                   ' Nombre de vies.
Public NbPomme As Byte                  ' Nombre de pommes mangées dans le niveau
Public Chrono As Double                 ' Permet de calculer la durée du jeu
Public Bonus As Double                  ' Compte à rebours pour gagner un bonus
Public T As Double                      ' Utilisée pour la gestion du temps
'-------------------------------------------------------------------------------------------------
Type TSerpent                           ' Type utilisateur : Serpent
    x As Integer                        ' Position X
    y As Integer                        ' Position Y
    Forme As String * 1                 ' Forme de l'élément à afficher
End Type
Public Serpent(0 To 200) As TSerpent    ' Nombre d'éléments, 0 représente la tête

'-------------------------------------------------------------------------------------------------
Sub InitialiseVariables()
'-------------------------------------------------------------------------------------------------
Niveau = 1
Points = 0
Vie = 3
Chrono = Timer
Range("FinDuJeu") = ""
Serpent(0).Forme = "0" ' Forme de la tête du serpent.
For i = 1 To UBound(Serpent)
    Serpent(i).Forme = "#" ' Forme de la queue du serpent.
Next i

End Sub
'-------------------------------------------------------------------------------------------------
```

Le module « Niveau » contient deux procédures. L'une pour initialiser un niveau, l'autre pour dessiner le niveau désiré :

```
'-------------------------------------------------------------------------------------------------
Sub InitialiseNiveau()
'-------------------------------------------------------------------------------------------------
' Efface l'intérieur du plateau :
Range("B2:AM39").ClearContents
Range("B2:AM39").Interior.ColorIndex = 15

' Affiche la tête du serpent :
MvX = 0: MvY = -1
Serpent(0).x = (EcranMaxiX + EcranMiniX) / 2
Serpent(0).y = EcranMaxiY + 2

' Affiche la queue du serpent :
TailleSerpent = AvanceParPomme
For i = 1 To TailleSerpent
    Serpent(i).x = 20
    Serpent(i).y = Serpent(i - 1).y + 1
Next i

' Affiche les informations :
NbPomme = 0
Range("Niveau") = Niveau
Range("Points") = Points
```

```
Range("Vie") = Vie
Range("NbPomme") = NbPomme

' Referme la porte :
Range("S1:U1") = "X"
Range("S1:U1").Interior.ColorIndex = 1
End Sub


'-------------------------------------------------------------------------------------------
Sub DessineNiveau()
'-------------------------------------------------------------------------------------------
Dim x As Byte, y As Byte, Couleur As Byte

If Niveau = 2 Then
    Range("H7:AH7") = "X"
End If

If Niveau = 3 Then
    Range("H7:AH7") = "X"
    Range("H17:AH17") = "X"
End If

If Niveau = 4 Then
    Range("H7:AH7") = "X"
    Range("H17:AH17") = "X"
    Range("H27:AH27") = "X"
End If

If Niveau = 5 Then
    Range("J8:J33") = "X"
    Range("AE8:AE33") = "X"
    Range("U8:U33") = "X"
End If

If Niveau = 6 Then
    Range("T5:T34") = "X"
    Range("F19:AI19") = "X"
End If

If Niveau = 7 Then
    Range("T5:T34") = "X"
    Range("F11:AI11") = "X"
    Range("F19:AI19") = "X"
    Range("F27:AI27") = "X"
End If

If Niveau = 8 Or Niveau = 9 Then
    Range("G8:AH8") = "X"
    Range("G8:G33") = "X"
    Range("G33:AH33") = "X"
    Range("AH8:AH33") = "X"
    Range("T8:U8") = ""
    Range("T33:U33") = ""
End If

If Niveau = 9 Then
    Range("T11:U30") = "X"
    Range("K20:AE21") = "X"
End If

If Niveau = 10 Or Niveau = 11 Then
    Range("D4:D36") = "X"
    Range("G7:G33") = "X"
    Range("J11:J30") = "X"
    Range("M15:M27") = "X"
    Range("P18:P24") = "X"
    Range("X18:X24") = "X"
    Range("AA15:AA27") = "X"
    Range("AE11:AE30") = "X"
    Range("AH7:AH33") = "X"
    Range("AK4:AK36") = "X"
```

```
    Range("T2:T39") = ""
    Range("B21:AM21") = ""
End If

If Niveau = 11 Then
    Range("D4:AK4") = "X"
    Range("G7:AH7") = "X"
    Range("J11:AE11") = "X"
    Range("M15:AA15") = "X"
    Range("P18:W18") = "X"
    Range("P24:W24") = "X"
    Range("M27:AA27") = "X"
    Range("J30:AE30") = "X"
    Range("G33:AH33") = "X"
    Range("D36:AK36") = "X"
    Range("T2:T39") = ""
    Range("B21:AM21") = ""
End If

If Niveau = 12 Then
    For i = 1 To 100
        x = 1 + Rnd() * (EcranMaxiX - 1)
        y = 1 + Rnd() * (EcranMaxiY - 1)
        If y > EcranMaxiY - 5 Then y = EcranMaxiY - 5
        Cells(y, x) = "X"
    Next i
End If

' Fin des niveaux :
If Niveau = 13 Then Vie = -1

' Met les X en noir et les vides en gris sur le plateau :
For x = EcranMiniX - 1 To EcranMaxiX + 2
    For y = EcranMiniY - 1 To EcranMaxiY + 2
    If Cells(y, x) = "" Then Couleur = 15 Else Couleur = 1
        Cells(y, x).Interior.ColorIndex = Couleur
    Next y
Next x

' Fait une pause de deux secondes :
Sleep 2000

' Pose une pomme :
Call PosePomme

End Sub
'-----------------------------------------------------------------------------------------------
```

Le module « Jeu » contient les diverses procédures du jeu.

Un bouton sur la feuille de calcul lance la procédure « Début » :

```
Option Explicit
Option Compare Text

'-----------------------------------------------------------------------------------------------
Sub Début()
'-----------------------------------------------------------------------------------------------
Call InitialiseVariables
Call InitialiseNiveau
Call DessineNiveau
Call DéplaceSerpent
End Sub

'-----------------------------------------------------------------------------------------------
Sub ChangeDirection(Direction As Integer)
'-----------------------------------------------------------------------------------------------
' Si gauche :
If Direction = -1 Then
    ' Si haut alors gauche :
```

```
    If MvX = 0 And MvY = -1 Then MvX = -1: MvY = 0: GoTo Fin
    ' Si gauche alors bas :
    If MvX = -1 And MvY = 0 Then MvX = 0: MvY = 1: GoTo Fin
    ' si bas alors droite :
    If MvX = 0 And MvY = 1 Then MvX = 1: MvY = 0: GoTo Fin
    ' Si droite alors haut :
    If MvX = 1 And MvY = 0 Then MvX = 0: MvY = -1: GoTo Fin
End If

' Si droite :
If Direction = 1 Then
    ' Si haut alors droite :
    If MvX = 0 And MvY = -1 Then MvX = 1: MvY = 0: GoTo Fin
    ' Si droite alors bas :
    If MvX = 1 And MvY = 0 Then MvX = 0: MvY = 1: GoTo Fin
    ' si bas alors gauche :
    If MvX = 0 And MvY = 1 Then MvX = -1: MvY = 0: GoTo Fin
    ' Si gauche alors haut :
    If MvX = -1 And MvY = 0 Then MvX = 0: MvY = -1: GoTo Fin
End If

Exit Sub

' Fait une pause pour éviter une nouvelle lecture trop rapide du clavier :
Fin:
Sleep 150

End Sub

'-----------------------------------------------------------------------------------------------
Sub DéplaceSerpent()
'-----------------------------------------------------------------------------------------------
' Traitement principal du jeu : pour déplacer le serpent, lire les interventions clavier,
' gérer les changements de direction. Boucle tant qu'il reste au moins une vie :
Do
    ' Masque le curseur :
    Range("BA50").Select

    ' Déplace la tête du serpent :
    Serpent(0).x = Serpent(0).x + MvX
    Serpent(0).y = Serpent(0).y + MvY

    ' Si le serpent rencontre un obstacle :
    If Cells(Serpent(0).y, Serpent(0).x) <> "" Then
        Call Obstacle ' Analyse le type d'obstacle
    End If

    ' Affiche la tête du serpent :
    Cells(Serpent(0).y, Serpent(0).x) = Serpent(0).Forme

    ' Lecture du clavier :
    Direction = 0
    T = Timer
    Do
        DoEvents
        If GetKeyState(vbKeyLeft) < 0 Then Direction = -1: Exit Do
        If GetKeyState(vbKeyRight) < 0 Then Direction = 1: Exit Do
        If GetKeyState(vbKeyControl) < 0 Then Sleep 50: Exit Do
        If GetKeyState(vbKeyDelete) < 0 Then Call TéléportationDuSerpent: Exit Do
        If GetKeyState(vbKeyEnd) < 0 Then Vie = 0
    Loop While T + 0.16 > Timer ' Durée de lecture à adapter au PC

    ' Fait l'éventuel changement de direction :
    Call ChangeDirection(Direction)

    ' Affiche la queue du serpent :
    Call AfficheQueue

    ' Affiche le temps passé :
    Range("Chrono") = (Timer - Chrono) / 100000

    ' Affiche le compte à rebours du bonus :
```

```
    If Bonus > 0 Then
        Range("Bonus") = (Bonus - Timer) / 100000
        If Range("Bonus") < 0 Then Bonus = 0: Range("Bonus") = ""
    End If

' Boucle tant qu'il reste au moins une vie au serpent :
Loop While Vie > 0

' Traitement pour la fin du jeu :
Call FinJeu

End Sub

'-----------------------------------------------------------------------------------------------
Sub AfficheQueue()
'-----------------------------------------------------------------------------------------------
' Efface le bout de la queue :
Cells(Serpent(TailleSerpent).y, Serpent(TailleSerpent).x) = ""
' Affiche la queue (élément précédent) :
For i = TailleSerpent To 1 Step -1
    Serpent(i).y = Serpent(i - 1).y
    Serpent(i).x = Serpent(i - 1).x
    Cells(Serpent(i).y, Serpent(i).x) = Serpent(i).Forme
Next i
' Ferme la porte d'où peut venir le serpent :
Range("T40") = "X"
End Sub

'-----------------------------------------------------------------------------------------------
Sub FinDuSerpent(AvecBonus As Boolean)
'-----------------------------------------------------------------------------------------------
' Efface un à un les éléments de la queue du serpent :
For i = TailleSerpent To 1 Step -1
    Cells(Serpent(i).y, Serpent(i).x) = ""
    Sleep 50
    ' Attribue un point par élément si c'est un changement de niveau :
    If AvecBonus = True Then
        Points = Points + 1
        Range("Points") = Points
    End If
Next i
End Sub

'-----------------------------------------------------------------------------------------------
Sub TéléportationDuSerpent()
'-----------------------------------------------------------------------------------------------
Dim x As Integer, y As Integer

' Efface le corps du serpent :
Call FinDuSerpent(False)

' Efface la tête du serpent :
Cells(Serpent(0).y, Serpent(0).x) = ""

' Donne une direction aléatoire pour compliquer la reprise en main :
Do
    MvX = Int(Rnd(1) * 3 - 1)
    MvY = Int(Rnd(1) * 3 - 1)
Loop While MvX <> 0 And MvY <> 0

' Recherche un nouvel emplacement vide pour poser le serpent :
Do
    Do
        x = Rnd() * EcranMaxiX
    Loop While x < EcranMiniX Or x > EcranMaxiX
    Do
        y = Rnd() * EcranMaxiY
    Loop While y < EcranMiniY Or x > EcranMaxiY
Loop While Cells(y, x) <> ""

' Initialise les positions de la tête et du corps :
For i = 0 To TailleSerpent
```

```
    Serpent(i).x = x
    Serpent(i).y = x
Next i
End Sub

'-------------------------------------------------------------------------------------------
Sub Obstacle()
'-------------------------------------------------------------------------------------------
' Si l'obstacle rencontré est une pomme :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If Cells(Serpent(0).y, Serpent(0).x) = FormePomme Then

    ' Fait grandir le serpent (sans rien afficher) :
    For i = TailleSerpent To TailleSerpent + AvanceParPomme
        Serpent(i).x = Serpent(TailleSerpent).x
        Serpent(i).y = Serpent(TailleSerpent).y
    Next i
    TailleSerpent = i - 1

    ' Donne des points et éventuellement un bonus :
    Points = Points + PointParPomme
    NbPomme = NbPomme + 1
    If Bonus > 0 Then Points = Points + Range("Bonus") * 100000
    Bonus = 0: Range("Bonus") = ""
    Range("Points") = Points
    Range("NbPomme") = NbPomme

    'Si le niveau est terminé, alors ouvre la porte :
    If NbPomme = NbPommeParNiveau Then
        Call OuvrePorte
    Else ' sinon  pose une nouvelle pomme :
        Call PosePomme
    End If

    Exit Sub
End If

' Si le serpent passe par la porte :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
If Cells(Serpent(0).y, Serpent(0).x) = FormePorte Then

    ' Efface le serpent en attribuant des points :
    Call FinDuSerpent(True)

    ' Bonus pour les vies restantes :
    Range("FinDuJeu") = "Bonus Vie"
    For i = 1 To Vie * 10
        Points = Points + 1
        Range("Points") = Points
        Sleep 100
    Next i
    Range("FinDuJeu") = ""

    ' Initialise puis dessine le niveau suivant :
    Niveau = Niveau + 1
    Vie = Vie + 1
    Call InitialiseNiveau
    Call DessineNiveau

    Exit Sub
End If

' Sinon, le serpent a rencontré un obstacle mortel (mur ou queue) :
' ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Vie = Vie - 1               ' Fait perdre une vie.
Call FinDuSerpent(False)    ' Efface le serpent sans attribuer de point.
Call InitialiseNiveau       ' Efface le plateau.
Call DessineNiveau          ' Redessine le niveau.

End Sub

'-------------------------------------------------------------------------------------------
```

```
Sub PosePomme()
'-------------------------------------------------------------------------------------------------
Dim x As Integer, y As Integer
' Recherche un emplacement vide dans le plateau :
Do
    Do
        x = Rnd() * EcranMaxiX
    Loop While x < EcranMiniX Or x > EcranMaxiX
    Do
        y = Rnd() * EcranMaxiY
    Loop While y < EcranMiniY Or x > EcranMaxiY
Loop While Cells(y, x) <> ""

' Affiche la pomme :
Cells(y, x) = FormePomme

' Relance le compte à rebours pour gagner un bonus :
Bonus = Timer + 12
End Sub

'-------------------------------------------------------------------------------------------------
Sub OuvrePorte()
'-------------------------------------------------------------------------------------------------
Range("S1:U1") = FormePorte
Range("S1:U1").Interior.ColorIndex = 6
End Sub

'-------------------------------------------------------------------------------------------------
Sub FinJeu()
'-------------------------------------------------------------------------------------------------
' Efface le plateau :
Range("B2:AM39").ClearContents
Range("B2:AM39").Interior.ColorIndex = 15

' Indique que c'est la fin du jeu :
Range("FinDuJeu") = "Fin du Jeu"
End Sub
```

Remerciements

Nous tenons à remercier **Pierre Fauconnier** et **Christophe** pour la relecture technique, **Winjerome** pour la mise au gabarit et **Claude Leloup** pour la relecture orthographique.