

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

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 pourrions 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 `Bonbonplus` 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.

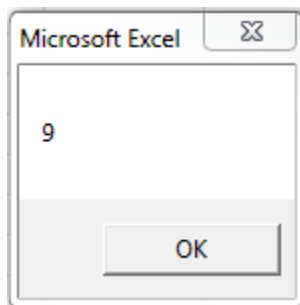
Revenons à notre programme : il nous reste encore à afficher, dans une fenêtre d'information, le contenu de la variable `Bonbon` à la fin du traitement en boucle, avec l'instruction **MsgBox**. Ne tenez pas compte pour le moment de l'info-bulle qui s'affiche quand vous saisissez cette instruction.

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 ».

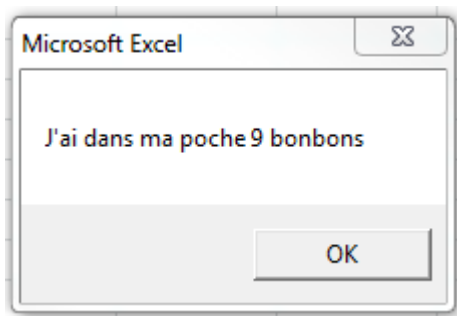
En plaçant la souris sur une variable, sa valeur s'affiche en info-bulle.

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 :

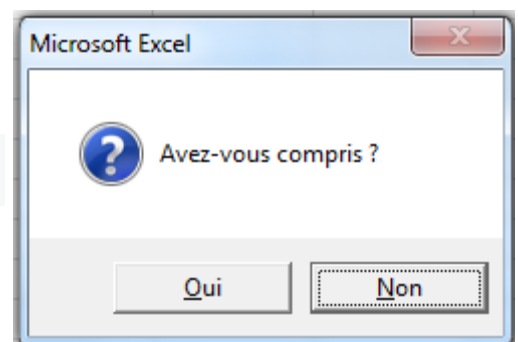
vbDefaultButton1	Le premier bouton est le bouton par défaut.
vbDefaultButton2	Le deuxième bouton est le bouton par défaut.
vbDefaultButton3	Le troisième bouton est le bouton par défaut.
vbDefaultButton4	Le quatrième bouton est le bouton par défaut.

- **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 +  
vbDefaultButton2
```

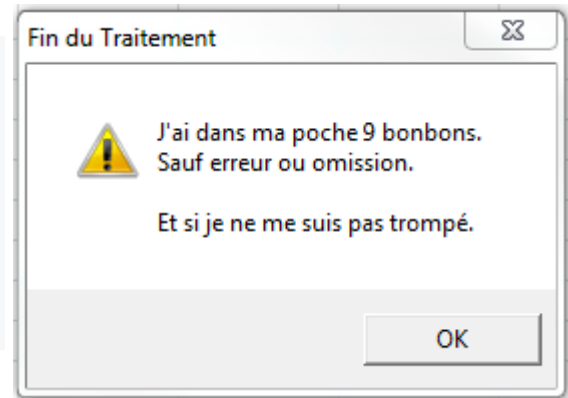


Bon à savoir : le caractère underscore (aussi appelé tiret bas, sous-tiret, blanc souligné, tiret du 8) permet de couper une ligne de code pour plus de lisibilité...

```
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 langages 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.

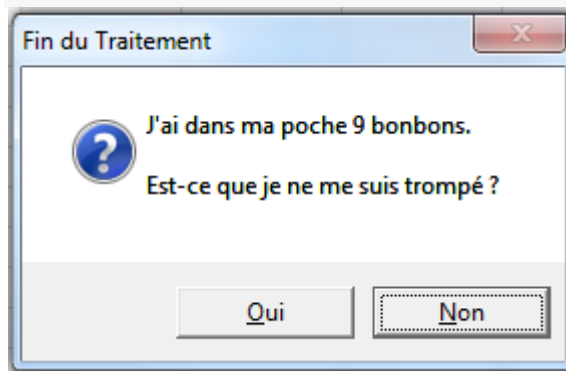
Dim Reponse As Long      ' Variable numérique pour le choix de
l'utilisateur.

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
    If Cells(Ligne, 1) = "x" Then Bonbon = Bonbon + 1 ' Faire...
    incrémente nombre bonbons, ' Si x alors
    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
    If Cells(Ligne, 1) <> "x" Then Exit For
x.
    Bonbon = Bonbon + 1
bonbons
Next 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 meilleure 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.

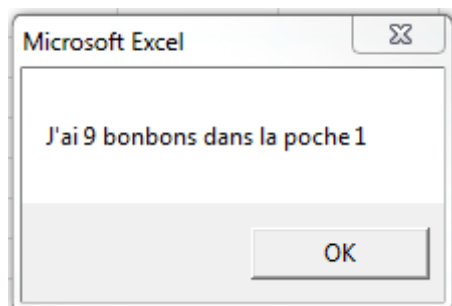
Bonbon = CompterPoche(1) ' Traitement sur la colonne 1.
MsgBox "J'ai " & Bonbon & " bonbons dans la poche 1"

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.

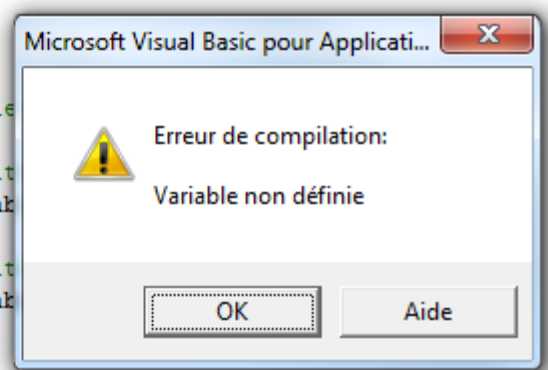
Dernière remarque au sujet de nos variables : elles ne sont pas identifiées par un code de couleur par l'éditeur, au contraire des mots clés du langage qui sont affichés en bleu. Donc il suffit d'une faute de frappe pour que la variable `Bombon` soit utilisée à la place de la variable `Bonbon` dans une 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) ' Traitement  
MsgBox "J'ai " & Bonbon & " bonbons dans la poche 1"  
  
Bonbon = CompterPoche(2) ' Traitement  
MsgBox "J'ai " & Bonbon & " bonbons dans la poche 2"  
  
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 stocker 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 devons 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'avons 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 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 scoubidoues. 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 avons 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 scoubidoues

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

MsgBox "J'ai " & MesBonbon & " bonbons et " _
    & MesScoubidou & " scoubidoues 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 1 ^{er} 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
    Rouge As Long
    Vert As Long
    Bleu As Long
End Type
```

' Déclaration d'un type de données personnel
' Définition du 1er élément
' Définition du 2e élément
' Définition du 3e élément
' 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 (**I**ndiceMini **To** **I**ndiceMaxi).

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 MémoireUnCarré()

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) ' Mémoire 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
Élément :	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
```

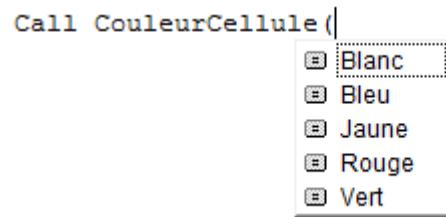
Puis nous créons une procédure « `CouleurCellule` » qui change la couleur de fond de la cellule active.

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)
```