JIM

JUNIOR **INVESTMENT**
MANAGEMENT by *TBS*

Formation VBA

Support de cours

2016 - 2017

JIM - Pôle Formation
Benjamin AUGEREAU, Laurent DELEBARRE, Pierre LE TREIZE

LCL
BANQUE ET ASSURANCE

SOMMAIRE

COURS 1 : INITIATION A EXCEL VBA

I. PRESENTATION DE VBA ET DES DIFFERENTES COMMANDES

A. QU'EST CE QUE VBA ?

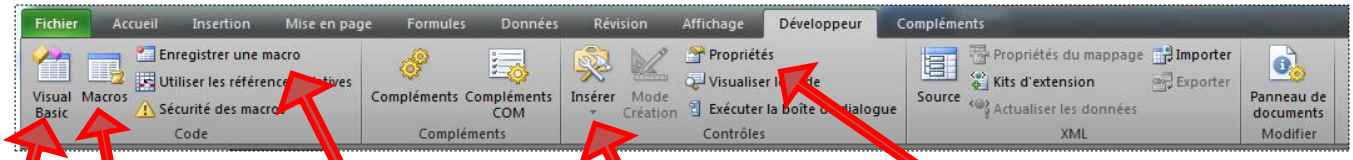
Le VBA (Visual Basic for Applications) est un langage de programmation qui nécessite une application hôte pour fonctionner. Dans notre cas, Excel remplira cette tâche. Mais sachez qu'il est également possible d'utiliser VBA sous Word. Ce langage va nous permettre d'automatiser des tâches sous Excel et de créer des fonctions. Les programmes qui permettent d'automatiser ces tâches sont appelées « macros ».

B. COMMENT ACCEDER A VBA ?

Onglet Fichier > Options > Personnaliser le ruban > Cocher « Développeur ».

C. PRESENTATION DE L'INTERFACE ET DES COMMANDES

1. PRESENTATION DE L'ONGLET « DEVELOPPEUR »



Bouton pour accéder et gérer les macros

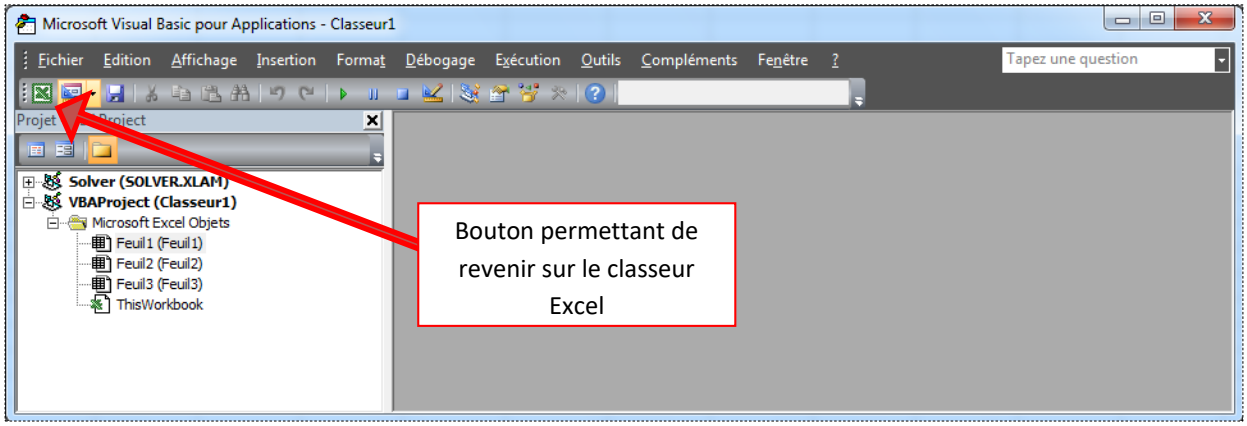
Bouton pour enregistrer une macro automatique

Permet d'accéder à l'éditeur Visual Basic

Permet d'insérer des éléments directement sur la feuille de calcul

Permet d'accéder aux propriétés de l'élément sélectionné

2. PRESENTATION DE L'EDITEUR VISUAL BASIC



Bouton permettant de revenir sur le classeur Excel

II. PRESENTATION DE L'ENREGISTREUR DE MACRO

Il existe deux façons de faire des macros sous VBA :

1) En utilisant l'enregistreur de macro.
2) En tapant directement le code.

Notons d'abord que la deuxième option est de loin la plus utilisée pour programmer en VBA. Cependant, nous allons nous intéresser à la première option ici : elle nous permet de créer une macro de manière « automatique ».

En appuyant sur le bouton « Enregistrer une macro » de l'onglet développeur on peut enregistrer une macro de manière empirique.

Il suffit d'appuyer sur ce bouton (vous pouvez éventuellement rentrer les caractéristiques de la macro) puis on peut commencer à effectuer les calculs et manipulations que vous souhaitez faire, enfin cliquer sur « Arrêter l'enregistrement ». Cette méthode enregistre ainsi ce que vous avez fait et le transforme en code. L'enregistreur de macro est finalement un simple magnétophone doté d'un traducteur « Actions > Code ».

A. EXEMPLE D'APPLICATION

Numérotez de 1 à 10 les 10 premières cases de la colonne A.
Numérotez de 11 à 20 les 10 premières cases de la colonne B.
Numérotez de 21 à 30 les 10 premières cases de la colonne C.
Numérotez de 31 à 40 les 10 premières cases de la colonne D.

Automatisons le processus suivant :

- Supprimer le contenu des colonnes A et C.
- Déplacer le contenu de la colonne B dans la colonne A.
- Déplacer le contenu de la colonne D dans la colonne C.

Pour ce faire, rien de plus simple que de lancer l'enregistrement de la macro en cliquant sur le bouton en question. Faites toutes les manipulations nécessaires puis finissez l'enregistrement en ré appuyant sur le bouton précédemment utilisé.

Allons ensuite dans l'éditeur Visual Basic (Raccourci Alt+F11) > Modules > Module 1. Dans l'éditeur Visual Basic, l'arborescence des fichiers apparait sur la gauche, sous la même forme que celle que l'on peut trouver sous Windows lorsque l'on navigue dans les dossiers : il vous suffit alors de double cliquer sur le module en question. Vous voyez le code de la macro que vous venez d'enregistrer.

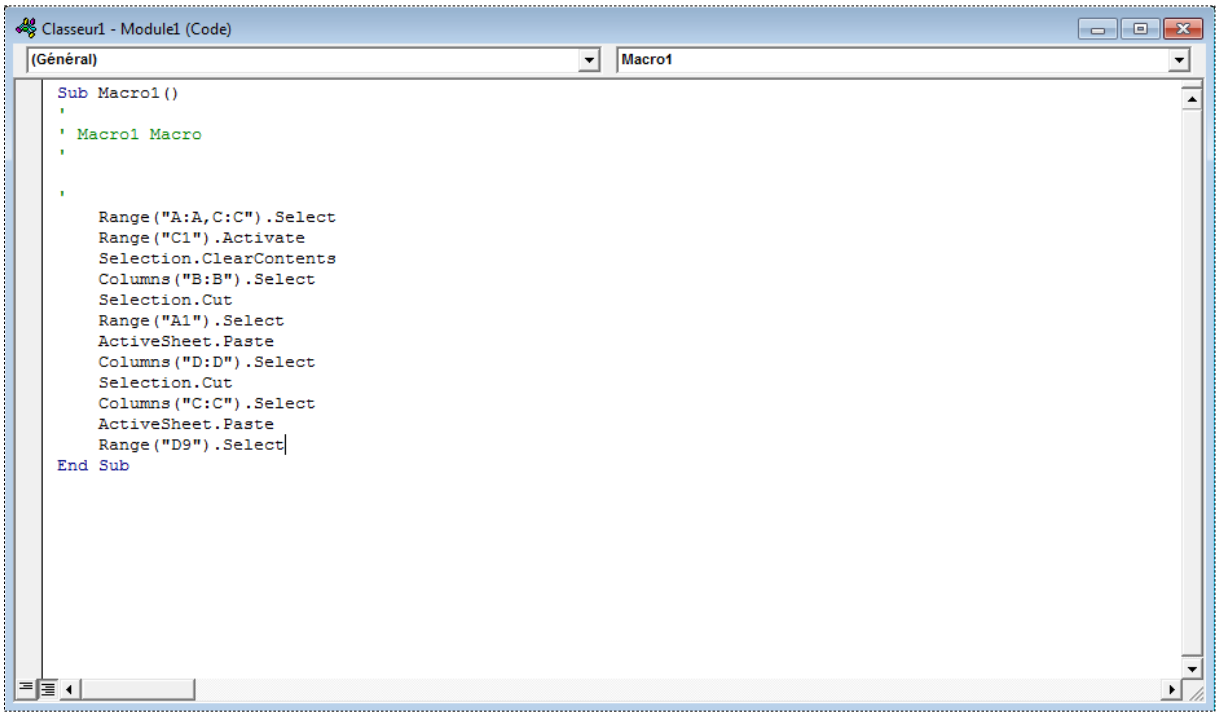
B. INTERETS DE L'ENREGISTREUR DE MACRO

Tout d'abord, celui-ci peut s'avérer très pratique si vous ave un besoin très simple : au lieu de passer par du codage, il vous suffit de faire un petit enregistrement, ce qui représente bien entendu un immense gain de temps. Mais cela n'est pas la fonction la plus intéressante de l'enregistreur de macro : comme vous pourrez le remarquer au fil de ces cours, il existe de nombreuses fonctions sous VBA ainsi que de nombreux objets et propriétés que nous serons amenés à manipuler. Il est évident qu'il n'est pas possible de tout retenir. L'enregistreur de macro peut ainsi vous sauver la mise. Si vous ne connaissez pas une fonction particulière, vous pouvez éventuellement la retrouver en enregistrant une macro puis en ouvrant le code créé pour en retirer l'expression qui vous intéresse. Un outil à ne pas sous-estimer donc !

III. PRESENTATION DU LANGAGE

Pour écrire une macro en tapant le code manuellement, il faut se rendre dans l'Editeur Visual Basic puis dans le menu « Insertion » et enfin cliquer sur « Module ». Vous pouvez également dans l'Editeur effectuer un clic droit sur le classeur ou une feuille du classeur puis suivre le même chemin que précédemment : Insertion > Module. Cela vous permet de créer un nouvel espace qui sera dédié aux macros que vous souhaitez créer. Il est également possible d'insérer les macros dans les diverses feuilles disponibles dans l'arborescence. Cependant, cela a une limite : une macro enregistrée dans une feuille ne sera utilisable que dans cette dernière. Le module vous permet donc de contourner ce problème en rendant la macro codé dans un module accessible à toutes les feuilles du classeur concerné.

A. LE PRINCIPE DES BALISES



Toute macro commence par « **Sub** » et finit par « **End Sub** ». Ici le nom de la macro est « **Macro1** ». Renommons là en « **manip_colonnes** » : attention, le titre d'une macro ne doit pas contenir d'espaces ! Tout le code qui sera rentré entre ces deux balises appartiendra à la macro « **manip_colonnes** ». Finalement, si l'on veut imager le principe des balises, on peut considérer que celles-ci forment des boîtes. Une fois la boîte ouverte, elle devra nécessairement être fermée si l'on veut pouvoir ensuite la manipuler. Vous le verrez, de très nombreux éléments sous VBA fonctionnent suivant ce principe. Nous vous conseillons donc de bien retenir cela : vous éviterez des oublis et cette image vous permettra de coder plus facilement en donnant un aspect plus visuel au code dans votre esprit.

B. LES COMMENTAIRES

Sur VBA, on peut ajouter un commentaire qui permet de mieux comprendre le code pour quiconque lirait votre travail ou tout simplement pour vous aider à vous repérer. Cela se fait en commençant la ligne par une apostrophe, ensuite on peut écrire ce que l'on souhaite (des indications utiles en général !). VBA sait alors qu'il faut ignorer cette partie du code. On note que la ligne devient verte. En effet, VBA colorise certaines parties du code afin de rendre sa lecture plus facile.

Astuce : cette technique est très pratique pour que VBA ignore une partie du code sans pour autant le supprimer.

C. LES BOUTONS

Nous allons insérer un bouton pour permettre de déclencher sur commande la macro. Pour insérer un bouton, vous devez sortir de l'Editeur Visual Basic et retourner sur la feuille Excel. Dans l'onglet « Développeur », vous allez cliquer sur « Insérer ». Dans « Contrôles de formulaires » sélectionnez la première option « Bouton ».

Dessiner votre bouton où vous voulez sur la feuille. Une fenêtre apparaît, sélectionner la macro que nous venons de renommer. Renumérotez les 10 premières cases de chacune des colonnes. Appuyez sur le bouton et admirez.

A noter : on peut également exécuter une macro en cliquant sur le bouton « Macros » (en haut à gauche de la barre des taches) et choisir la macro que l'on veut exécuter puis cliquer sur « Exécuter ». De plus, il est également possible d'assigner une macro à tout élément disponible dans l'onglet « Insertion » de votre classeur Excel : une forme quelconque (flèche, smiley, …) peut donc se voir assignée une macro.

IV. OBJETS ET METHODES

A. DEFINITIONS : OBJETS ET COLLECTIONS D'OBJETS

- **Les objets** : tout ce que nous manipulons sous VBA est un objet (une cellule, un bouton, une feuille, un classeur, un graphique, etc.). Par exemple, le terme utilisé pour désigner une cellule ou une plage de cellule est « **Range** ». Pour une ligne, on utilisera « **Row** ».

- **Les collections d'objets** : les objets qui ont des points communs sont regroupés dans des « collections d'objets ». Par exemple, l'ensemble des feuilles du classeur ouvert sont regroupées dans « **Worksheets** ». Autre exemple de collections : « **ChartObjects** » regroupe tous les graphiques. Tous les objets au sein d'une même collection sont indexés à partir de 1. Si l'on considère l'exemple de la collection « Worksheets », la feuille 1 correspondra à « **Worksheets(1)** », la feuille 2 correspondra à « **Worksheets(2)** » et ainsi de suite. Si le nom de la feuille 1 est « Feuil1 », il est également possible d'appeler cette feuille en utilisant « **Worksheets(« Feuil1 »)** ». Nous allons revenir sur ces notations dans le point suivant.

A noter : les noms des collections finissent toujours par un « **s** » à la différence des objets simples. De plus, elles peuvent nous permettre de modifier les paramètres de tous les objets d'une même collection. Par exemple :

- « **WorkSheets.Add** » permet d'ajouter une nouvelle feuille.
- « **ChartObjects.ChartArea.Shadow = True** » permet de rajouter une ombre sous tous les graphiques du cours.

B. LES METHODES

Les méthodes vont nous permettre de manipuler les objets et les collections d'objets. Elles remplissent donc un rôle très important puisqu'elles nous offrent la possibilité de travailler sur les objets. Pour mieux expliquer leur fonctionnement et la manière dont nous les employons, nous allons prendre pour exemple deux méthodes très utiles : « **Select** » et « **Activate** ».

1. LA METHODE « SELECT »

La méthode « Select » permet de sélectionner des objets :

- Pour sélectionner la cellule A6, on va écrire : « **Range("A6").Select** ».
- Pour sélectionner les cellules A1 à A4 on va écrire : « **Range("A1 :A4").Select** ».
- Pour sélectionner les cellules A1 et A4 : « **Range("A1 ,A4").Select** ».
- Pour sélectionner la colonne A : « **Range("A:A").Select** » ou « **Columns("A").Select** ».
- Pour sélectionner les colonnes A à C : « **Range("A:A:C:C").Select** » ou « **Columns("A :C").Select** ».
- Pour sélectionner les colonnes A et C : « **Range("A:A,C:C").Select** ».
- Pour sélectionner une plage de cellules renommée en « cellules » : « **Range("cellules").Select** ».

Cet exemple permet d'aborder la syntaxe : nous allons toujours commencer par citer l'objet sur lequel nous souhaitons travailler. Ensuite nous mettons un « **.** » qui fait en quelque sorte office de lien entre l'objet et la méthode que nous allons lui appliquer. Après ce « **.** », nous inscrivons par conséquent la méthode. Cela marche tout le temps de cette manière quand on s'attache aux méthodes.

Bien retenir la syntaxe : « **Objet.Méthode** ».

A noter : le « **:** » signifie qu'on prend toutes les cases entre la première et la dernière indiquées, la virgule ne prend que les cases mentionnées.

Méthode très utile liée à la sélection des cases :

« **.Offset(X,Y)** » : cette méthode permet de décaler la sélection de la cellule active de X lignes et de Y colonnes.

2. LA METHODE « ACTIVATE »

La méthode « Activate » permet de rendre actif un objet.

Supposons que je veuille rendre active la quatrième feuille de mon classeur, je vais écrire dans mon code : « **Worksheets(4).Activate** ». On peut également appeler la même en écrivant « **Worksheets("Feuil4")** » si le nom de la feuille 4 n'a pas été changé. Cela nous renvoie aux notations présentées précédemment. Cette méthode est très importante. Pourquoi ?

Imaginons une situation très simple : vous êtes sur la feuille 1, des données y sont inscrites, et après traitement par une macro, vous aimeriez qu'un graphique soit construit à partir de ces données et ensuite placé dans la feuille 2. Si vous n'utilisez pas la méthode « **Activate** », l'objet « Graphique » va par défaut s'ajouter à la feuille active qui par exemple peut être la feuille 1. Cela ne répond donc pas à votre besoin. L'une des méthodes nous permettant de palier à ce problème est donc d'utiliser la méthode « **Activate** » afin de désigner la feuille qui devient active et ainsi où le contenu devra être placé. Avantage supplémentaire : cela va vous envoyer directement au résultat une fois la macro exécutée. Dans cet exemple, si la méthode « **Activate** » est utilisé à bon escient, vous atterrirez sur la feuille 2 une fois la macro exécutée.

Une manière d'illustrer ce problème est de travailler sur cet autre exemple :

```
Sub test()
        Worksheets(1).Activate    'nous activons la feuille 1

        If Range("A1") > 3 Then   'nous testons le contenu de la cellule A1 de la feuille 1 (= la feuille active)
        Worksheets(2).Activate    'si la condition est remplie, alors on active la feuille 2
        Range("A1") = 6           'et on y inscrit la valeur 6 dans la cellule A1
        Else
        Worksheets(3).Activate    'sinon on active la feuille 3
        Range("A1") = "feuille3"  'et on y inscrit "feuille3" dans la cellule A1
        End If

End Sub
```

En reproduisant ce code, vous observerez par vous-même le résultat selon la valeur que vous inscrirez dans la cellule A1 de la feuille n°1.

Cependant, vous le comprenez cette solution est assez lourde. En effet, cela vous demande d'activer une feuille à chaque fois que vous désirez travailler sur un objet contenu par celle-ci. Heureusement, il existe également une autre solution nous permettant d'échapper à l'utilisation de la méthode « **Activate** » (qui n'en demeure pas moins une méthode très importante et à retenir).

Pour présenter cette seconde solution, nous allons la comparer à une recherche d'adresse (au sens géographique). Imaginons que vous souhaitiez vous rendre en salle 301 dans le bâtiment Alaric de l'ESC Toulouse. Très schématiquement, vous allez devoir d'abord rentrer dans l'ESC Toulouse puis vous rendre au bâtiment Alaric puis vous diriger jusqu'à la salle 301. Cette petite histoire représente parfaitement l'environnement de programmation VBA : les salles et les bâtiments sont des collections d'objets tandis que la salle 301 est un objet tout comme le bâtiment Alaric ou l'ESC Toulouse.

En VBA, cette adresse va s'écrire : « **ESCs(« ESC_Toulouse »).Bâtiments(« Alaric »).Salles(« 301 »)** ». Et tout a son importance dans cette syntaxe. Vous remarquez les « **s** » à « **ESCs** », « **Bâtiments** » et « **Salles** » : on considère les collections par conséquent. Ensuite, entre parenthèses et entre guillemets, on précise au sein de cette collection l'objet qui nous intéresse. Et on affine petit à petit jusqu'à obtenir l'objet qui nous intéresse. Le fonctionnement sous VBA est exactement le même : on va reprendre ce principe d'adresse pour travailler sur des objets situés à divers endroits.

Par exemple, si je veux travailler sur la cellule A1 de la feuille 2, je vais l'appeler de la manière suivante : « **Worksheets(2).Range(« A1 »)** ». Vous remarquerez cependant que « **Range** » ne prend pas de « **s** », c'est une des petites nuances qui peut survenir dans VBA. Il s'agit donc bien d'un objet, malgré une syntaxe plus longue. Cela ne modifie donc en rien ce que nous avons pu dire sur la syntaxe des méthodes : cela reste « **Objet.Méthode** », à la « différence » que « **Objet** » va s'écrire de manière plus complète.

Pour illustrer ce point, on peut réutiliser l'exemple du code donné précédemment en utilisant cette fois la syntaxe que l'on vient de voir :

```
Sub test2()

    If Worksheets(1).Range("A1") > 3 Then     'la condition s'intéresse à la cellule A1 de la feuille n°1
    Worksheets(2).Range("A1") = 6             'si elle est réalisée, elle modifiera la cellule A1 de la feuille n°2
    Else
    Worksheets(3).Range("A1") = "feuille3"    'sinon elle modifiera la cellule A1 de la feuille n°3
    End If

End Sub
```

Reproduisez ce code et observez le résultat.

Notre conseil : gardez toujours à l'esprit cette métaphore de l'adresse géographique et utilisez de manière judicieuse la méthode « **Activate** ». Elle est par exemple très utile pour afficher à l'utilisateur le résultat de la macro.

3. AUTRES EXEMPLES DE METHODES

Autres exemples :

```
.Value = "TBS Finance"          'Donne la valeur « TBS Finance » à un objet
.ClearContents                  'Efface le contenu de l'objet : dans le cas d'une cellule, efface son contenu par ex.
```

On peut en cumuler plusieurs à la suite :

```
.Font.Size = 78                 'Modifier la taille de la police
.Font.Bold = True/False         'Mettre en gras ou non la police
.Font.Italic = True/False       'Mettre en italique ou non la police
.Font.Underline = True/False    'Souligner ou non la police
.Font.Name = "Arial"            'Changer de police d'écriture
.Borders.Value = 1              'Pas de bordure = 0
```

A noter :

- Ce qui est à gauche du signe égal va prendre la « valeur » de ce qui est à droite du signe égal.
- Il y a une multitude de méthodes, vous pouvez les trouver via l'aide d'Excel VBA.
- Mettre « **False** » n'est pas forcément très utile, mieux vaut utiliser le « **'** » du commentaire si l'on veut que VBA ignore la ligne de code.

C. FONCTIONNALITE « WITH »

Supposons que je veuille modifier plusieurs caractéristiques du même objet, comme la taille de la police, la police d'écriture en elle-même ou les bordures. Je vais alors utiliser la fonction « **With** » qui va me permettre d'utiliser plusieurs méthodes à un même objet :

```
With Range("A6")
        .Font.Size = 78
        .Font.Bold = True
        .Font.Italic = True
        .Font.Underline = True
        .Font.Name = "Arial"
        .Borders.Value = 1
End With
```

On peut même limiter encore la répétition de « **Font** » :

```
With Range("A6")
        .Borders.Value = 1
        With .Font
                .Size = 78
                .Bold = True
                .Italic = True
                .Underline = True
                .Name = "Arial"
        End With
End With
```

V. PRESENTATION DES VARIABLES

Les variables permettent de stocker des données. Il faut voir les variables comme de l'espace de stockage. L'erreur à ne pas commettre est d'associer en permanence variable et cellule d'une feuille Excel. En effet, VBA peut très bien se passer complètement du contenu des cellules et pourtant travailler avec des variables.

Imaginons par exemple un formulaire demandant à l'utilisateur des valeurs : le formulaire pourra stocker ces valeurs dans des variables puis VBA pourra travailler sur ces variables et finalement renvoyer un graphique ou un résultat. Pendant tout ce traitement, le classeur Excel n'aura pas servi. Il n'y a donc pas nécessairement de lien direct à établir entre variables et cellules du classeur.

Pour utiliser une variable, la déclaration n'est pas obligatoire sous VBA. Par défaut, si vous utilisez une nouvelle variable sans l'avoir déclarée, elle prendra la valeur 0 et sera de type « **Variant** ». Le tableau situé sur la page suivante récapitule les types disponibles sous VBA.

Pourquoi est-ce important de déclarer le type de la variable ? Nous verrons plus tard que cela permet de faire des tests pour savoir si la variable est de bon type avant même de commencer à manipuler la variable. En effet, certaines fonctions ne peuvent fonctionner qu'avec un type de variable. De plus, il existe derrière la déclaration du type un souci d'optimisation : en effet, le type « Variant » demande une allocation de mémoire plus importante que le type « Integer » ce qui en soi est tout à fait logique. Si toutes les variables sont déclarées sans type de variable ou si elles ne sont pas déclarées, elles vont toutes être de type « Variant » et par conséquent, beaucoup de mémoire va être allouée aux

variables ce qui ralentira l'exécution du programme qui bénéficiera ipso facto de moins de mémoire à sa disposition pour effectuer les traitements.

Nom	Type	Détails
Byte	*Numérique*	Nombre entier de 0 à 255.
Integer	*Numérique*	Nombre entier de -32'768 à 32'767.
Long	*Numérique*	Nombre entier de - 2'147'483'648 à 2'147'483'647.
Currency	*Numérique*	Nombre à décimale fixe de -922'337'203'685'477.5808 à 922'337'203'685'477.5807.
Single	*Numérique*	Nombre à virgule flottante de -3.402823E38 à 3.402823E38.
Double	*Numérique*	Nombre à virgule flottante de -1.79769313486232D308 à 1.79769313486232D308.
String	*Texte*	Texte.
Date	*Date*	Date et heure.
Boolean	*Boolean*	True (vrai) ou False (faux).
Object	*Objet*	Objet Microsoft.
Variant	*Tous*	Tout type de données (type par défaut si la variable n'est pas déclarée).

Comment déclarer le type d'une variable ?

- Il peut être préférable de déclarer le type de la variable au tout début de votre programme par souci de clarté.
- Il faut écrire : « **Dim Ma_variable As Type_de_variable** » en remplaçant « **ma_variable** » par le nom que vous voulez donner à votre variable et « **Type_de_variable** » par le type que vous voulez donner à votre variable.
- Exemple : « **Dim Chapeau As String** ».

A noter : si l'on veut utiliser une variable dans plusieurs macros différentes, il faut la déclarer à l'extérieur des macros, dans le même module. Par exemple :

```
Dim chapeau As String

----------

Sub macro1

'La variable chapeau peut être utilisée ici.

End sub

----------

Sub macro2

'La variable chapeau peut également être utilisée ici.

End sub
```

Attention : la valeur de la variable reste dans la variable à la fin de la procédure.

VI. PRESENTATION DE LA FONCTIONNALITE « MSGBOX »

« **MsgBox** » : il s'agit d'une fonctionnalité très pratique que l'on développera de manière plus poussée au cours 4 (Les boites de dialogue et formulaires).

Dans n'importe quelle macro on peut utiliser la fonctionnalité « **MsgBox** », elle a pour but de diffuser un message à l'utilisateur. Par exemple:

```
Sub test_msgbox1
        MsgBox ("TBS Finance is the best association!")
End sub
```

On peut également afficher le contenu d'une variable à l'écran grâce à la fonction « **MsgBox** » en séparant la variable des zones de texte par « **&** » :

```
Sub test_msgbox2
        MsgBox ("TBS Finance is the best association!")
        Dim nombre As Integer
        nombre = 23
        MsgBox ("Et elle compte"& nombre &" membres")      'Texte affiché à l'écran : Et elle compte 23 membres
End sub
```

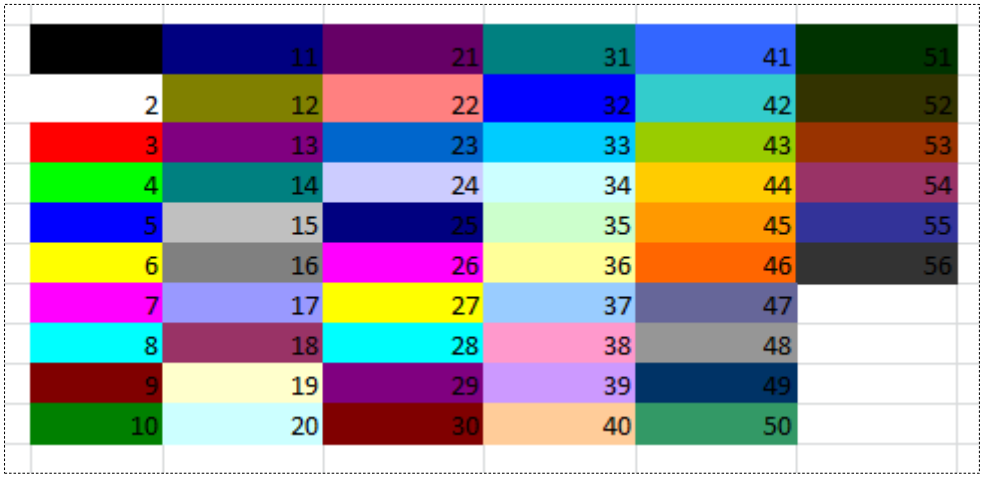
VII. LES METHODES LIEES AUX COULEURS

Comme on l'a déjà vu avec la méthode « **Font** », VBA peut servir à mettre en forme. Par exemple, il est possible de mettre de la couleur. On peut changer la couleur de fond d'une cellule, ou même la couleur de l'onglet de chacune des feuilles.

Il y a deux façons de mettre de la couleur sous Excel VBA :

A. LA METHODE « COLORINDEX »

La méthode « **ColorIndex** » comprend une palette de 56 couleurs :



	11	21	31	41	51
2	12	22	32	42	52
3	13	23	33	43	53
4	14	24	34	44	54
5	15	25	35	45	55
6	16	26	36	46	56
7	17	27	37	47	
8	18	28	38	48	
9	19	29	39	49	
10	20	30	40	50	

Exemples d'utilisation :

- Pour changer la couleur de fond d'une case : « **Range("A1").Interior.ColorIndex = 1** ».
- Pour changer la couleur de police d'une case : « **Range("A1").Font.ColorIndex = 45** ».
- Pour changer la couleur de l'onglet d'une feuille : « **Sheets("Feuil1").Tab.ColorIndex = 7** ».

Un des problèmes de cette méthode est qu'elle est limitée à 56 couleurs.

B. LA METHODE « COLOR »

Cette méthode permet de mettre toutes les couleurs que l'on veut. Elle fonctionne sur le principe du code de couleurs RGB. Sa syntaxe est la suivante :

```
.Color = (R,G,B)    'où R, G et B représentent les valeurs de la couleur.
```

Pour avoir ces valeurs, ouvrez Paint puis cliquez sur « Modifier les couleurs » dans l'onglet Accueil. Choisissez votre couleur, il faut alors prendre les nombres des cases ROUGE, VERT, BLEU.

Par exemple : ma couleur sous Paint a pour valeurs « Rouge = 250, Vert = 100, Bleu = 50 » alors je vais coder en VBA selon les exemples suivants :

- Pour changer la couleur de fond d'une case : « **Range("A1").Interior.Color = (250,100,50)** ».
- Pour changer la couleur de police d'une case : « **Range("A1").Font.Color = (250,100,50)** ».
- Pour changer la couleur de l'onglet d'une feuille : « **Sheets("Feuil1").Tab.Color = (250,100,50)** ».

COURS 2 : LES CONDITIONS ET LES BOUCLES

I. LA CONDITION « IF »

A. PRESENTATION DE LA STRUCTURE

La condition « **If** » prend la structure suivante dans le code VBA :

```
If (condition) Then
        'Instructions si condition remplie
Else
        'Instructions si condition non remplie
End If
```

B. PRESENTATION DES OPERATEURS DE CONDITIONS POSSIBLES

La plupart du temps nous allons utiliser une condition avec une comparaison. Il nous faut donc présenter les principaux outils de comparaison :

Condition que l'on veut coder	**Code VBA correspondant**
si X est égal à Y alors	**IF X = Y THEN**
si X est différent de Y alors	**IF X <> Y THEN**
si X est plus petit que Y alors	**IF X < Y THEN**
si X est plus grand que Y alors	**IF X > Y THEN**
si X est plus petit ou égal à Y alors	**IF X <= Y THEN**
si X est plus grand ou égal à Y alors	**IF X >= Y THEN**

Il existe également d'autres opérateurs de conditions possibles pour cumuler certaines conditions. Par exemple:

Condition à coder	**Code VBA correspondant**	**Explications**
Si X est égal à Y et à Z	**IF X=Y AND X=Z THEN ...**	Cumul de 2 conditions
Si X est égal à Y ou à Z	**IF X=Y OR X=Z THEN ...**	Au moins l'une des 2 conditions est réalisée
Si X n'est pas égal à Y	**IF NOT X=Y THEN ...**	La condition n'est pas réalisée (elle est fausse)

II. LA FONCTION « ELSEIF »

La fonction « **ElseIf** » permet d'enchaîner les conditions. Elle prend la structure suivante :

```
If (condition n°1) Then
        'Instructions si condition n°1 remplie
ElseIf (condition n°2) Then
        'Instructions si condition n°2 remplie
ElseIf (condition n°3) Then
        'Instructions si condition n°3 remplie
ElseIf (condition n°4) Then
        'Instructions si condition n°4 remplie
Else
        'Instructions si aucune des conditions n°1, n°2, n°3 et n°4 ne sont remplies
End If
```

Commentaires : Si la condition 1 n'est pas remplie alors le code VBA va tester la deuxième condition, si elle n'est pas remplie, il va tester la troisième, etc. Si aucune des conditions n'est remplie il exécutera alors les conditions qui se situent après « **Else** ».

III. LA CONDITION « SELECT CASE »

Si l'on enchaîne beaucoup de « **ElseIf** » avec des conditions différentes, on peut utiliser une autre boucle qui est la boucle « **Select Case** **»**. Elle a la syntaxe suivante :

```
Select Case (objet à tester)
Case Is (condition 1)
        'Instructions si le test 1 est réussi
Case Is (condition 2)
        'Instructions si le test 2 est réussi
Case Is (condition 3)
        'Instructions si le test 3 est réussi
End Select
```

Cette boucle va tester successivement les différentes conditions.

A noter : avec cette boucle, on peut utiliser les autres opérateurs de comparaison :

Condition à coder	CODE VBA
Si la case à tester doit avoir une valeur > X	**CASE IS > X**
Si la case à tester doit avoir une valeur < X	**CASE IS < X**
Si la case à tester doit avoir une valeur différente de X	**CASE IS <> X**
Si la case à tester doit avoir une valeur supérieure ou égale à X	**CASE IS >= X**
Si la case à tester doit avoir une valeur inférieure ou égale à X	**CASE IS <= X**

A noter : avec cette boucle, on peut comparer directement la valeur avec plusieurs valeurs :

Condition à coder	CODE VBA
Si la case à tester doit avoir une valeur > X et Y	**CASE IS > X,Y**
Si la case à tester doit avoir une valeur inférieure ou égale à X et Y	**CASE IS <= X,Y**
ETC.	
Si la case à tester doit avoir une valeur comprise entre 6 et 10	**CASE 6 TO 10**

IV. CONDITIONS EN FONCTION D'UN TYPE

Nous n'avons mis que des exemples de comparaisons numériques dans les présentations des conditions précédentes. Mais il peut être utile de savoir que l'on peut également tester le type d'un objet ou d'une variable :

3 exemples :

- **« IsNumeric(X) = True »** permet de tester si X est bien de type numérique.
- **« IsDate(X) = True »** permet de tester si X a bien la forme d'une date.
- **« IsEmpty(X) = True »** permet de tester si X est vide.

A. EXEMPLE AVEC LA FONCTION ISNUMERIC

La fonctionnalité « **IsNumeric** » permet de tester le contenu d'un objet VBA ou d'une variable pour savoir s'il est de type numérique.

Exemple : tester le contenu de la case A1 et le multiplier par 2 s'il est numérique, afficher un message d'erreur s'il n'est pas numérique:

```
If IsNumeric (Cells(1,1)) = True Then
        Cells(1,2).Value = Cells(1,1)*2
Else
        Cells(1,2).Value= "NOT NUMERICAL!"
End If
```

A noter : le « **= True** » n'est pas obligatoire, en revanche si vous voulez que X ait n'importe quel format sauf un format numérique, il faudra absolument écrire :

```
If IsNumeric(X)= False Then
        'Instructions si X n'est pas de type numérique
End If
```

Attention : une situation particulière peut apparaitre. Dans le premier exemple, imaginons que la cellule concernée par le test soit vide. L'utilisateur considère tout naturellement qu'il ne s'agit donc pas d'un type numérique. Et pourtant, si on exécute la macro, le « **NOT NUMERICAL !** » ne va pas être renvoyé. Au lieu de cela, on voit un « **0** » apparaitre dans la cellule B1. Pourquoi ? On l'a dit tout à l'heure, si vous ne déclarez pas une variable, par défaut VBA va lui assigner le type « **Variant** » et la valeur nulle. L'idée est la même pour un objet vide. Dans notre cas, la cellule vide va être considérée

comme de type « **Variant** » donc potentiellement numérique (puisque le type « **Variant** » accepte aussi bien les chiffres que les chaînes de caractères) et aura pour valeur « **0** » dans le calcul effectué une fois la condition vérifiée.

Pour illustrer cela, regardons ce petit exemple qui nous montre que cela est également valable pour les variables :

```
Sub test5()

        Dim Y As Variant
        ' Y = 3
        ' Y = "mot"

        If IsNumeric(Y) Then
        MsgBox ("c'est numérique")
        Else
        MsgBox ("ce n'est pas numérique")
        End If

End Sub
```

Regardons d'un peu plus près cet exemple : trois possibilités s'offrent à nous.

- Le code reste tel qu'il est : on déclare donc la variable en type « **Variant** » et on ne lui assigne aucune valeur. Par conséquent, lors du test, nous allons avoir la première boîte de dialogue « **c'est numérique** » étant donné que le type « **Variant** » comprend les valeurs numériques.
- Deuxième possibilité : on déclare la variable en « **Variant** » et ensuite on lui assigne la valeur « **3** ». Dans ce cas là, elle est devenue numérique. Par conséquent, nous allons encore avoir la boîte de dialogue « **c'est numérique** ». Mais cette fois-ci, cela sera plus justifié d'une certaine manière étant donné qu'il sera parfaitement clair que la variable est bien numérique.
- Dernière possibilité : nous déclarons la variable en « **Variant** » et ensuite nous lui assignons la valeur « **mot** ». A ce moment, le test va nous renvoyer la boîte de dialogue « **ce n'est pas numérique** ».

Cela vous permettra sans aucun doute d'éviter bien des situations d'incompréhension face aux résultats que pourraient donner vos macros.

B. CAS PARTICULIER : CONDITIONS EN FONCTION D'UN TYPE DE VARIABLE

On peut également exécuter des instructions en fonction du type d'une variable. La nuance parait faible mais est pourtant de taille : précédemment, les conditions se faisaient indifféremment en fonction du type de l'objet ou de la variable. Ici, on ne va s'attacher qu'aux variables. Pour se restreindre au type de la variable, nous devons introduire une nouvelle fonction « **VarType** ». En voici la structure :

```
If VarType(ma_variable) = VBtype_de_variable Then …
```

Exemple : « **If VarType(X)= vbinteger Then …** »

Les différents types de variable en question ont déjà été vus durant l'initiation.

C. LA FONCTION « IS NOTHING »: INTRODUCTION AUX VARIABLES OBJETS

Dans le cours d'introduction, nous vous avons présenté différents types de variable. Il s'agissait pour la quasi-totalité de variables recevant des valeurs numériques ou des chaines de caractères. Pour présenter la fonction « **Is Nothing** », nous

allons cependant devoir vous présenter de nouvelles variables : **les variables objets**. Le plus simple pour vous les présenter est de vous donner une déclaration de variable objet :

```
Dim X As Range
```

Qu'est ce que cela signifie ? Tout simplement, nous avons déclaré une variable « **X** » de type « **Range** » ou en français « **Plage de cellules** ». Or nous l'avons vu, « Range » désigne habituellement l'objet. Ici, cette syntaxe en fait un type de variable. La variable « **X** » prendra donc comme valeur des plages de cellule.

Par exemple :

```
Dim X As Range
Set X = Range(« A1 :B12 »)
```

On remarque une petite nuance par rapport aux variables classiques : l'apparition du « **Set** » qui est indispensable pour assigner une valeur à une variable objet.

Présentation de la fonction « Is Nothing » :

Tout d'abord, voice un petit exemple :

```
Sub test4()

        Dim X As Range                  'On déclare une variable objet
        Set X = Range("A1:B8")          'On lui assigne une valeur

        If X Is Nothing Then            'On effectue un test sur cette variable : on vérifie si elle est vide
        MsgBox ("vide")
        Else
        MsgBox ("non vide")
        End If

End Sub
```

La fonction « **Is Nothing** » permet de vérifier l'état de la variable : a-t-elle reçu une valeur ou non ? Si la variable n'a été que déclarée, la condition sera remplie et par conséquent la macro va nous renvoyer une boîte de dialogue « **vide** ». Dans le cas contraire (celui de notre exemple), la variable ayant reçu une valeur, elle n'est pas vide et par conséquent, la macro nous renvoie une boîte de dialogue « **non vide** ».

Une question subsiste : pourquoi vous avoir présenté les variables objets ? Il existe une raison toute simple à cela : la fonction « Is Nothing » ne peut être appliquée qu'aux variables objets. Si vous tentez de faire ce test sur une variable de type classique, VBA va vous renvoyer une erreur d'incompatibilité de type.

V. CONDITIONS EN UTILISANT LA FONCTION « LIKE »

Avec VBA, on peut également comparer deux chaînes de caractères, ce qui peut constituer une condition. Pour cela, on peut utiliser la fonction « **Like** ».

Attention : après « **Like** » il faut toujours mettre la chaîne de caractères entre guillemets.

Attention bis : VBA est sensible à la casse (minuscules et majuscules) ! Pour qu'il n'y soit pas sensible, il faut mettre « **Option Compare Text** **»** en début de macro ou de module.

* pour remplacer aucun ou plusieurs caractères **quelconques.**
? pour remplacer UN caractère **quelconque.**
pour remplacer UN caractère **numérique** de 0 à 9.

On peut également utiliser des plages de caractères :

- [xyz] peut remplacer x, y ou z
- [x-z] peut remplacer x, y ou z
- [148] peut remplacer 1, 4 ou 8
- [1-3] peut remplacer 1, 2 ou 3

Attention : ces plages de caractère ne peuvent remplacer QU'UN caractère !

Astuce : [*?#] peut remplacer « * », « ? » ou « # ».

Par exemple : supposons que l'on ait déclaré « **ma_variable = Bobby 31** »

```
If ma_variable Like "*obb*" Then
        'Instructions qui vont se lancées car la condition est remplie
End If
```

```
If ma_variable Like "Bob?y 31" Then
        'Instructions qui vont se lancées car la condition est remplie
End If
```

```
If ma_variable Like "Bobby 3#" Then
        'Instructions qui vont se lancées car la condition est remplie
End If
```

```
If ma_variable Like "[ABC]obby 3[123]" Then
        'Instructions qui vont se lancées car la condition est remplie
End If
```

Remarque : on peut évidemment cumuler plusieurs symboles.

```
If ma_variable Like "*b?y 3#" Then
        'Instructions qui vont se lancées car la condition est remplie car * remplace "Bo", ? remplace b et # remplace 1
End If
```

Présentation de « ! » dans une chaîne de caractères entre crochets :

Si on met « **!** » au début d'une chaîne de caractères, cela remplace n'importe quel symbole, autre que ceux précisés entre crochets.

Par exemple:

```
If ma_variable Like "[!DEF]obby 3[!456]" Then
```

```
        'Instructions qui vont se lancées car la condition est remplie
End If
```

VI. LA BOUCLE « WHILE »

Les boucles « **While** » exécutent un bloc d'instruction tant que la condition est vraie.

Elles ont la structure suivante :

```
While (condition)
        'Instructions si la condition est réalisée
Wend
```

Par exemple, on peut faire une petite boucle infinie:

```
While  Date <> Date +1
        Workbooks.Add
        MsgBox ("Un nouveau classeur a été ajouté")
Wend
```

Commentaires : la fonction « Date » est disponible dans VBA et renvoie la date du jour. Par conséquent, la condition « **Date <> Date + 1** » est toujours remplie, ce qui explique pourquoi il s'agit d'une boucle infinie. Ensuite vous trouvez « **Workbooks.Add** » : la méthode « **Add** » permet de créer un nouveau classeur. Cette méthode peut vous être particulièrement utile dans un classeur si vous l'utilisez pour ajouter de nouvelles feuilles par exemple.

A noter : la boucle suivante existe également, même si elle reprend exactement le même principe que la boucle « **While** » :

```
Do While (condition)
        'Instructions tant que la condition est réalisée
Loop
```

Le seul avantage de cette boucle est que l'on peut utiliser la fonctionnalité « **Exit Do** » avec alors que ce n'est pas possible avec une simple boucle « **While Wend** » (fonctionnalité que l'on verra un peu plus loin dans ce cours).

VII. ALTERNATIVE A LA BOUCLE « WHILE » : LA BOUCLE « LOOP WHILE »

La boucle « **Loop While** » fonctionne de la même façon que la boucle « **While** » à la seule condition que les instructions vont être exécutées une fois, quoi qu'il arrive. En effet, la condition arrive à la fin de la boucle :

```
Do
        'Instructions réalisées la première fois puis les fois suivantes SI LA CONDITION EST RÉALISÉE'
Loop While (condition)
```

Attention : pour que la boucle ne soit pas infinie et qu'elle s'arrête, il faut qu'un élément de la condition soit modifié dans la condition!

VIII. LA BOUCLE « DO UNTIL »

Les boucles « **Do Until** » exécutent un bloc d'instruction jusqu'à ce que la condition soit atteinte. Il faut donc qu'un élément de la condition soit modifié dans la boucle pour que celle-ci ne soit pas infinie !

Cette boucle a la structure suivante :

```
Do Until (condition)
        'Instructions si la condition n'est pas réalisée
Loop
```

IX. LA BOUCLE « FOR » ET LA FONCTION « STEP »

La boucle « **For** » permet de répéter un nombre de fois défini un bloc d'instructions. Elle utilise une variable qui est incrémentée (croissante) ou décrémentée (décroissante).

Vous pouvez voir la structure sur les exemples suivants :

```
For k = 1 To 10    'Incrémente de 1 en 1
'Instructions
Next
```

```
For k = 0 To 10 Step 2 'Incrémente de 2 en 2 (nombres pairs)
' Instructions
Next
```

```
For k = 10 To 1 Step  - 1 'Décrémente de 1 en 1
' Instructions
Next
```

A noter : le « **k** » est une variable à part entière qui peut être utilisée dans les instructions.

X. LA BOUCLE « FOR EACH … IN »

Cette fonction, à la différence de la boucle « **For** », permet de faire une boucle avec des objets d'une collection ou des éléments d'un tableau.

```
For Each Objet In Collection
'Instructions
Next
```

Un exemple illustrant la boucle « **For Each … In** » :

```
Sub afficher_noms()

        Dim feuille As Worksheet          'On declare une variable objet de type Worksheet

        For Each feuille In Worksheets    'On parcourt chaque feuille de la collection Worksheets
                MsgBox (feuille.Name)     'On affiche le nom de chaque feuille grâce à la méthode Name
        Next feuille

End Sub
```

Cette procédure nous permet donc d'afficher chaque nom de feuille dans une « **MsgBox** ». Nous utilisons donc ici la possibilité qui nous est offerte de parcourir tous les éléments d'une collection. Mais comme nous l'avons dit plus haut, il est également possible de parcourir tous les éléments d'un tableau présent sous une feuille Excel. Comment faire ?

Par exemple, utiliser la fonction « **Alea** » pour générer un tableau avec des chiffres compris entre 0 et 20. Pour ce faire, dans la cellule « A1 », vous allez rentrer la formule suivante : « **= ALEA () * 20** ». Cette formule permet de générer un nombre aléatoire compris entre 0 et 20. Vous copiez ensuite cette formule dans toute une plage de cellules, la plage « **A1 :L14** » par exemple. Ensuite créer une macro avec ces instructions et observez le résultat !

```
Sub test()

        Dim Variable As Range, Variable2 As Range
        Set Variable = Range("A1:L14")

        For Each Variable2 In Variable
                If Variable2 < 10 Then
                        Variable2.Font.Color = RGB(255, 0, 0)
                Else
                        Variable2.Font.Color = RGB(0, 255, 0)
                End If
        Next Variable2
End Sub
```

Passons aux explications : nous commençons par déclarer deux variables objet qui sont toutes les deux de type « **Range** ». Ensuite, nous assignons à « **Variable** » une plage de cellules, en l'occurrence celle que nous avons remplie de valeurs aléatoires comprises entre 0 et 20. La manière la plus simple de voir les choses pour mieux comprendre ce qui va suivre est de considérer que cette assignation fait de « **Variable** » une collection d'objets, qui eux ne sont rien d'autre que les cellules de cette plage.

Ensuite, nous rentrons dans la boucle « **Each For … In** » : le fonctionnement et l'explication sont les mêmes que pour le premier exemple que nous vous avions présenté. La ligne signifie donc « Pour chaque cellule de la plage ». Ensuite, on rentre dans des conditions « **If** ». Si la cellule considérée a une valeur inférieure à 10, elle deviendra rouge. Sinon elle deviendra verte. Et on finit par boucler avec le « **Next Variable2** ».

Si cela vous parait toujours mystérieux, il vous suffit de faire un changement de nom simple qui rendra la macro sans doute plus lisible : transformer « **Variable** » en « **Plage** » et « **Variable2** » en « **Cellule** ». Cela vous donnera alors l'instruction : « **For Each Cellule In Plage** ».

XI. LA FONCTION « EXIT » (EXIT DO, EXIT FOR, EXIT SUB ET EXIT FUNCTION)

Pour quitter prématurément une boucle, on peut utiliser la fonction « **Exit** ». Plusieurs fonctions en découlent :

- « **Exit Do** » pour sortir d'une boucle « **Do Until** » ou « **Loop While** ».
- « **Exit Sub** » pour sortir de la procédure (qui n'est pas obligatoirement dans une boucle).
- « **Exit Function** » pour sortir d'une fonction (qui n'est pas obligatoirement dans une boucle, nous verrons les fonctions dans un autre cours).

Pour quitter prématurément une boucle, la solution optimale est de mettre un « **If** » à l'intérieur de la boucle qui, si la condition est réalisée, va lancer l'instruction « **Exit** ». Par exemple:

```
For i = 1 To 10
        'Instructions
        If (condition) Then
        Exit For
        End If
Next
```

XII. LA FONCTION « MOD »

La fonctionnalité « **Mod** » permet de donner le reste d'une division. Sa structure est la suivante : « **X Mod Y** » renvoie le reste de la division de X par Y.

Exemple d'utilisation avec les nombres pairs ou impairs :
Un utilisateur rentre en A1 un nombre entier et on veut savoir s'il est pair ou impair :

```
Sub parité()
        Dim nombre As Integer

        If nombre Mod 2 = 0 Then
                MsgBox ("Le nombre "& nombre & " est pair.")
        Else
                MsgBox("Le nombre " & nombre & "est impair.")
        End If
End Sub
```

XIII. APPLICATION : CREER UN JEU DE CALCUL MENTAL

A. PRINCIPE DU JEU

Nous allons créer un jeu très simple faisant appel aux capacités de calcul mental du joueur. Le principe est très simple : nous demandons au début du jeu combien de parties le joueur veut faire puis le jeu se lance. En quoi consiste une partie ? L'ordinateur va générer un nombre entier compris entre 1 et 50 et l'utilisateur va devoir en donner le carré. A la fin de la partie, l'ordinateur va donner au joueur son score.

On remarque donc ici que le jeu est composé de 3 phases qu'il va falloir retranscrire en code :

- Interroger le joueur sur le nombre de parties qu'il désire faire.
- Lancer les parties.
- Donner le score.

Comme vous allez pouvoir le constater, ce petit jeu va faire intervenir un grand nombre d'éléments que nous avons pu voir dans ce cours et dans le cours d'introduction. Essayez de trouver vous-même la solution avant de regarder la correction !

B. LA SOLUTION

```
Sub jeu_carre ()

Do      'Ouverture de la boucle principale

        Dim N As Integer            'Déclaration de la variable nombre_de_manches
        N = InputBox(" Combien de parties voulez vous faire ? ")      'On assigne à cette variable le nb desiré par le
                                                                       joueur

        Dim s As Integer            'Déclaration de la variable qui comptera le nombre de bonnes réponses
        s = 0                       'Initialiser à 0 : indispensable

                For k = 1 To N      'Boucle permettant d'exécuter le nombre_de_manches

                        X = Fix(Rnd * 50)           'Génération du nombre aléatoire : Fix permet d'obtenir un entier

                        If InputBox("Quel est le carré de " & X & " ?") = X ^ 2 Then    'Test de la réponse donnée
                        MsgBox (" VRAI ")
                        s = s + 1           'Compteur des bonnes réponses
                        Else
                        MsgBox ("FAUX")
                        End If

                Next k

        MsgBox ("Votre score est : " & s & " / " & N)                   'Affichage du score
        R = InputBox(" Voulez-vous rejouer ? Oui = 1 / Non = 0")        'On demande à l'utilisateur s'il veut continuer

Loop While R = 1        'Fin de la boucle principale : on teste la réponse de l'utilisateur contenue dans la variable R

End Sub
```

COURS 3 : PROCEDURES ET FONCTIONS

I. INTRODUCTION

Toutes les macros que nous avons créées jusqu'à maintenant sont des **procédures**. Il peut être utile de savoir que l'on peut **appeler des procédures depuis une autre procédure en l'appelant simplement par son nom** ! On peut écrire une procédure soit dans les modules de VBA, soit dans les onglets représentant chaque feuille.

Pour insérer un module, on va dans VBA (ALT+F11) > Clic droit sur le nom de votre fichier dans l'explorateur à gauche > Insertion > Module. Le module est l'endroit où vous allez pouvoir taper le code de vos procédures et fonctions.

Par exemple, pour appeler une procédure depuis une autre procédure, on écrira dans le même module :

```
Sub premiere_procedure ()
'Instructions de votre première procédure : faites le test avec une MsgBox par exemple.
End Sub
----------
Sub deuxieme_procedure()
'Appel de la première procédure :
premiere_procedure
End Sub
```

Commentaire : en exécutant « **deuxieme_procedure** », VBA va alors lancer « **premiere_procedure** ».

A. PRESENTATION DE OPTION EXPLICIT

Cette option, une fois écrite au début de votre programme, va **vous obliger à déclarer le type de vos variables**. En effet, si vos variables ne sont pas déclarées alors qu' « **Option Explicit** » est présent, alors VBA affichera un message d'erreur pour toute variable non déclarée qu'il rencontrera dans votre macro.

Pour l'insérer automatiquement au début de chacun de vos modules, vous pouvez faire, toujours dans VBA : Outils > Options > Cocher l'option « Déclaration des variables obligatoire ».

Pour rappel, déclarer ses variables est important principalement pour deux raisons :

1) La lisibilité de votre programme.
2) Le gain d'espace mémoire : si une mémoire n'est pas déclarée, alors elle prend le type Variant qui est beaucoup plus consommateur de mémoire. L'exécution de votre programme en est donc ralentie !

B. PRESENTATION DU PUBLIC/PRIVATE

Il existe deux types deux procédures : celles accessibles depuis n'importe quelle partie du fichier Excel (« **Public Sub** ») et celles qui ont un accès plus restreint (« **Private Sub** »). Pour le moment, toutes les procédures que nous avons créées étaient publiques : elles commençaient par « **Sub** » et finissaient par « **End Sub** ».

Pour commencer une procédure privée, il faudra commencer par « **Private Sub** » et finir par « **End Sub** ».

A noter : on ne peut pas lancer une « **Private Sub** » directement depuis un bouton sur une feuille Excel. Les « **Private Sub** » sont en particulier réservés aux procédures que l'on appelle depuis une autre procédure.

II. LES ARGUMENTS

A. PRESENTATION GENERALE

Les arguments permettent de donner la valeur d'un ou plusieurs paramètres nécessaires à l'exécution d'une procédure. Ils se mettent entre les deux parenthèses qui suivent directement le nom de votre procédure. Prenons un exemple : supposons que dans le même module j'écrive les deux procédures suivantes :

```
Private Sub racine_cubique (nombre As Double)
MsgBox (" La racine cubique de : " & nombre & " est : " & Nombre ^ (1/3))
End Sub

----------

Sub Test_Racine_Cubique ()

        Nombre = InputBox ("Entrez un nombre positif")      'L'utilisateur rentre un nombre

        If IsNumeric (nombre) = False Then                  'On teste la valeur pour savoir si elle est numérique
                MsgBox("Valeur non numérique!")
        ElseIf nombre < 0 Then                              'Si elle est bien numérique, on teste son signe
                MsgBox ("Le nombre ne remplit pas les conditions!")
        Else                                                'Si toutes les conditions sont remplies, on appelle
                racine_cubique(nombre)                      'la Private Sub racine_cubique pour traiter le nombre
        End If


End Sub
```

Dans « **Private Sub racine_cubique(nombre As Double)** » , « **nombre** » **est l'argument de la procédure « racine_cubique ».** On le déclare en écrivant « **As Double** », ce qui signifie qu'il est de type « **Double** ». Les types d'argument sont exactement les mêmes que les types de variables *(Voir cours 1 : initiation).*

A noter : il n'est pas obligatoire de déclarer le type d'argument. Cependant, pour les mêmes raisons que pour la déclaration des variables, il est judicieux de déclarer les arguments en donnant leur type.

A noter : une procédure peut contenir plusieurs arguments, il suffit juste de les séparer par une virgule quand on les déclare. Par exemple :

```
Private Sub declaration (texte As String, age, nombre As Integer)
```

Dans cet exemple, les arguments « **age** » et « **nombre** » sont déclarés comme des nombres entiers.

B. LES ARGUMENTS OPTIONNELS : PRESENTATION DE LA FONCTIONNALITE ISMISSING

Tous les arguments que nous avons déclarés jusqu'à présent étaient obligatoires : si l'utilisateur ne met pas d'argument quand il appelle la procédure alors cela va boguer. **En effet, les arguments sont obligatoires par défaut.**

En revanche, il est possible de mettre des **arguments optionnels** dans la déclaration des arguments. Pour cela il faut faire **précéder** votre argument de l'opérateur « **Optional** ». Par exemple :

```
Private Sub declaration(texte As String, Optional age As Integer, Optional nombre As Double)
```

En particulier, on peut tester la présence d'un argument optionnel avec l'opérateur « **IsMissing** » qui fonctionne de la même façon que « **IsNumeric**» ou « **IsDate** ». Cependant, l'utilisation de « IsMissing » est un peu particulière : en effet, « **IsMissing** » ne peut être utilisé qu'avec un argument optionnel de type « **Variant** ».

Exemple : avec une fonction (fonctionnement des fonctions détaillé dans la partie suivante)

```
Function affichage(nom As String, Optional age As Variant)

        If IsMissing(age) = True Then
                MsgBox (affichage = nom)
        Else
                MsgBox (affichage = nom & " a " & age & " ans.")
        End If
End Function

----------

Sub test()
        Call affichage("Laurent")  // Call affichage("Laurent", 35)      'Utiliser une solution puis l'autre
End Sub
```

Constatation : En lançant la macro, on s'aperçoit que suivant la solution sélectionnée et donc suivant les arguments rentrés, le contenu de la boîte de dialogue varie. Si dans la « **Function** », vous changez le type de l'argument optionnel « **age** », vous allez pouvoir constater qu'en utilisant « **Call affichage(« Laurent »)** », la boîte de dialogue va vous donner « **Laurent a 0 ans** ». Cela illustre bien ce que nous avons pu dire précédemment : seul le type « **Variant** » fonctionne avec l'opérateur « **IsMissing** ».

III. LES FONCTIONS SOUS VBA

A. LES FONCTIONS PREDEFINIES

- « **Date** » : Renvoie la date actuelle.
- « **Len** » : Compte le nombre de caractère dans une chaîne de texte.
- « **Fix** » : Renvoie la partie entière d'un nombre.
- « **Int** » **:** Renvoie la partie entière d'un nombre + 1 (entier directement supérieur au nombre).
- « **Typename** » : Donne le type d'objet.
- « **Shell** » : Exécute une application.
- « **Rnd** » : Renvoie un chiffre entre 0 et 1. **Astuce** : 10*Rnd => renvoie un nombre entre 0 et 10.
- « **Max** »**/** « **Min** » : Renvoie la valeur maximum/minimum.

Astuce : Pour explorer les différentes fonctions il faut aller dans le VBA, puis appuyez sur « Explorateur d'objet » (ou F2) et allez dans « **Worksheet.Function** ».

A noter : pour avoir un exemple d'utilisation, il vous suffit de regarder l'application du cours n°2.

B. LES FONCTIONS PERSONNALISEES :

A la différence des procédures « Sub » qui peuvent finalement ne rien renvoyer ou renvoyer plusieurs résultats, les fonctions « Function » ne peuvent renvoyer qu'une seule valeur. Elles vont cependant nous être très utiles pour créer de véritables fonctions personnalisées sous Excel. En effet, les fonctions « **Function** » sous VBA fonctionnent de la même façon que les fonctions classiques que vous utilisez dans Excel sur votre feuille de calculs (*quand vous tapez « = » dans une case de votre classeur pour y coller une fonction*). Par exemple cette fonction va nous permettre de calculer la racine cubique d'un nombre :

```
Function RacineCubique (nombre As Double)
        RacineCubique = nombre ^ (1/3)
End Function
```

La fonction ainsi créée peut être utilisée comme une fonction classique sous Excel. On peut donc désormais taper « **=** » dans une case de notre classeur et commencer à taper « **RacineCubique** » pour voir apparaître dans la liste des fonctions disponibles notre fonction VBA que nous venons de créer. En d'autres termes, si on tape dans une cellule « **=RacineCubique** », Excel reconnaitra la fonction que nous venons de créer.

A noter : une fonction ne peut pas être enregistrée à partir de l'enregistreur de macro. Elle doit être codée à la main.

De plus, on peut également appeler une fonction depuis n'importe quelle procédure. Reprenons l'exemple de la racine cubique, nous écrirons ce texte dans un module :

```
Function RacineCubique(nombre As Variant)
        RacineCubique = nombre ^ (1 / 3)
End Function

---------------------------------------------------------------------------------------------------------------------------------

Sub test_racine_cubique()

        Dim nombre1 As Variant
        nombre1 = InputBox ("Entrez un nombre positif")

        If IsNumeric(nombre1) = False Then
                MsgBox ("Valeur non numérique!")
        ElseIf nombre1 < 0 Then
                MsgBox ("Le nombre ne remplit pas les conditions!")
        Else
                MsgBox ("La racine cubique de " & nombre1 & " est : " & RacineCubique (nombre1))
        End If

End Sub
```

Une petite différence apparait par rapport à notre premier exemple de la racine cubique : nous avions déclaré « **nombre** » comme étant de type « **Double** » alors qu'ici nous utilisons le type « **Variant** ». Pourquoi ? Dans la « **Sub** », nous utilisons un test de comparaison avec « **IsNumeric** » pour vérifier si l'utilisateur rentre bien une valeur numérique pour commencer. Mais si jamais il rentre un caractère comme une lettre par exemple, le test « **IsNumeric** » nous permettra de lui afficher une « **MsgBox** » pour lui indiquer son erreur.

Cependant, si vous déclarez « **nombre** » et « **nombre1** » en « **Double** », le test ne fonctionnera pas si jamais l'utilisateur rentre une lettre et un bug va apparaitre. Et cela pour une raison toute simple : le type « Double » ne peut recevoir que des informations numériques. Par conséquent, si on tente d'assigner à une variable de type « **Double** » un élément non numérique, le programme va s'arrêter. Faites le test pour vérifier.

Par conséquent, on se doit d'utiliser le type « **Variant** ». Pour « **nombre1** », cela doit vous paraitre évident après cette explication. On peut pourtant se poser la question pour l'argument « **nombre** ». Vous pouvez à nouveau faire un test : déclarer « **nombre** » comme de type « **Double** » et « **nombre1** » en « **Variant** ». Et à nouveau, message d'erreur. L'explication de ce bug est assez complexe, mais la manière la plus simple de présenter cela est de faire référence à une chose toute bête que l'on voit en grammaire : la concordance des temps. De la même manière qu'il faut respecter les temps dans une phrase, il faut respecter les types de variables entre les différentes procédures et fonctions, ou sinon vous vous exposez à des messages d'erreur difficiles à interpréter.

A noter : une fonction qui a des arguments ne peut pas être lancée directement depuis une feuille Excel via un bouton par exemple.

C. BYREF/BYVAL

Devant chaque argument déclaré d'une procédure ou d'une fonction on peut écrire : « **ByRef** » et « **ByVal** ». Ces opérateurs permettent de ne modifier que les variables nécessaires dans les fonctions.

Si on utilise « **ByRef** » devant un argument alors la fonction ou procédure modifiera le contenu de la variable argument (si la fonction est amenée à la modifier).
Si on utilise « **ByVal** » devant un argument alors la fonction ou procédure ne modifiera pas le contenu de la variable argument (si la fonction est amenée à la modifier), elle utilisera juste la valeur pour effectuer les instructions.

A noter : si on n'écrit rien, c'est « **ByRef** » qui est appliqué par défaut à votre variable.

Exemple :

```
Function racine_cubique(ByVal nombre As Double)
        nombre = nombre ^ (1 / 3)
        racine_cubique = nombre
End Function
-----------------------------------------------------------------------------------------------------------------------------------------------

Sub testracine()
        Dim nombre1 As Double
        nombre1 = InputBox("Ecris un nombre :")
        MsgBox ("La racine cubique de " & nombre1 & " est: " & racine_cubique(nombre1))
End Sub
```

Commentaire : nous voyons donc que la fonction fonctionne parfaitement. Essayons maintenant le même programme en enlevant « **ByVal** », c'est donc « **ByRef** » qui s'applique par défaut. Et là surprise, « **nombre1** » a changé de valeur en passant dans la fonction racine_cubique.

Ici, on voit donc très bien l'utilité de « **ByRef** » et « **ByVal** » si l'on veut préserver la valeur initiale rentrée par l'utilisateur.

D. APPLICATION.VOLATILE ET APPLICATION.CALCULATE

1. APPLICATION.VOLATILE

Cette instruction sert à rendre une fonction volatile, i.e. lorsqu'une modification sera faite dans une feuille, la fonction va automatiquement se relancer et être recalculée. En effet, la fonction va se recalculer à chaque modification d'une valeur de n'importe quelle cellule, de n'importe quelle feuille et de n'importe quel classeur ouvert.

Pour qu'une fonction soit volatile, il suffit de placer comme suit dans votre fonction :

```
Function volatile(arguments)
Application.Volatile
'instructions
End Function
```

Commentaire : Ensuite, quand vous utilisez cette fonction dans une case de votre feuille Excel, si vous modifiez l'un des arguments, la fonction va se recalculer automatiquement.

2. APPLICATION.CALCULATE

« Application.Calculate » sert à relancer les calculs dans le classeur au moment où vous entrez cette instruction dans le fil des instructions dans une fonction. Elle diffère de « **Application.Volatile** » puisqu'une fonction volatile se recalcule automatiquement à l'inverse de « **Application.Calculate** » est une instruction qui va se lancer uniquement quand VBA atteindra cette instruction dans le fil des instructions de la fonction.

IV. RAPPELS UTILES

A. PETIT RAPPEL POUR NOMMER UNE PROCEDURE SUB OU FUNCTION

Tous les caractères ne sont pas permis. Voici une liste des caractères que VBA ne tolère pas :

- Le nom de la procédure ne peut pas commencer par un chiffre.
- Les espaces et les points sont interdits.
- Le langage VBA ne fait pas la différence entre les majuscules et les minuscules.
- Les caractères suivants sont interdits dans un nom : # $ % & @ ^ * !
- **Attention** : quand vous écrivez le nom d'une procédure « **Function** » assurez-vous que celle-ci ne soit pas l'homonyme d'une cellule (par exemple A1).
- Un nom ne peut excéder 255 caractères.

B. RAPPEL SUR LES METHODES ET PROLONGEMENT

1. LA METHODE SELECT

Cette méthode permet de sélectionner un élément. Par exemple :

- « **Range(« A1 »).Select** » permet de sélectionner la cellule A1.
- « **Columns (“A:C”).Select** » permet de sélectionner les colonnes A, B, C.
- « **Rows (“1:5”).Select** » permet de sélectionner les lignes 1 à 5.

2. LA METHODE OFFSET

« **Range(“A1”).Offset(1,2).Select** » : cette ligne de code permet de sélectionner la cellule qui se trouve une ligne en dessous de A1 et deux colonnes à droite, donc de sélectionner la cellule C2.

3. DIFFERENCE ENTRE LES METHODES TEXT ET VALUE

« **Text** » va renvoyer l’intégralité d’une chaine de caractères alors que « **Value** » ne renverra que la valeur numérique.

Exemple : mettez 12,3 € dans la cellule A1. Et codez dans un nouveau module :

```
Sub test ()
        Msgbox (Range(« A1 »).Value)
        Msgbox (Range(« A1 »).Text)
End Sub
```

Lancez la procédure pour voir la différence !

4. LA METHODE COUNT

Cette propriété permet de compter le nombre d’éléments.

```
Sub test()

        MsgBox (Sheets.Count)

End Sub
```

Cet exemple permet de compter le nombre de feuilles dans votre classeur et de donner le résultat à l’utilisateur.

5. LA METHODE ADDRESS

Cette propriété permet de renvoyer l’adresse d’un objet « **Range** », « **Column** », « **Cells** » ou « **Row** » en notation absolue (avec les dollars). Par exemple, essayez cette macro :

```
Sub Test()
        MsgBox (Range("A1").Address)
End Sub
```

6. LA METHODE HASFORMULA

Cette méthode retourne « **True** » si la cellule contient une formule ou « **False** » si la cellule ne contient pas de formule.

7. LA METHODE FONT

Cette méthode permet de modifier la police d'une cellule ou d'une plage : on peut alors mettre le texte en gras, en italique, changer de police pour mettre le texte en Arial ou autre, etc. Par exemple, on peut trouver ce type de code : « **Range(« A1 »).Font.Bold = True** ». Le texte de la cellule A1 sera alors en gras.

La méthode « **Interior** » permet de mettre en forme le fonds de la cellule : « **Range("A1").Interior.ColorIndex = 3** ».

8. LA METHODE FORMULA

Cette méthode permet d'écrire une formule dans une cellule. Par exemple :

```
Sub test()
        Range("A1").Formula = " = somme (A1 :A4) "
End Sub
```

9. LA METHODE NUMBERFORMAT

Cette méthode permet de changer le format du contenu d'une cellule.
Par exemple, la macro suivante permet de mettre sous forme de pourcentage le contenu de la colonne A :

```
Sub mettreenpourcentage()
        Columns("A:A").NumberFormat = "0,00%"
End Sub
```

V. EXEMPLE SORTI DU COURS D'ADVANCED CORPORATE FINANCE

La macro suivante permet de calculer la valeur d'une entreprise suivant la méthode de Gordon Shapiro :

```
Sub Gordon_Shapiro()

        Dividende = InputBox("Dividende ? ")
        Taux_de_croissance = InputBox("Taux de croissance ?")
        Cout_du_capital = InputBox ("Cout du capital ?")

        Valorisation = (Dividende / (Cout_du_capital - Taux_de_croissance))
        Valorisation1 = Fix(Valorisation)

        MsgBox ("La valorisation de l'entreprise est de " & Valorisation1 & " € ")

End Sub
```

COURS 4 : LES BOITES DE DIALOGUE ET LES FORMULAIRES

I. LES « MSGBOX »

Nous avons déjà utilisé cette fonction dans les cours précédent. Toutefois il est intéressant d'expliquer concrètement comment elle fonctionne.

« MsgBox » permet de poser une question, et de créer des boutons pour y répondre. Nous pouvons également y incorporer un titre. La structure est la suivante :

« **MsgBox (« question que vous voulez poser », boutons, « titre de la boîte de dialogue »)** ».

Pour les boutons, voici les possibilités :

- « **vbYesNo** » : Affiche le bouton Oui et Non
- « **vbOkOnly** » : Affiche le bouton Ok
- « **vbOkCancel** » : Affiche le bouton Ok et Annuler
- « **vbArbortRetryIgnore** » : Affiche le bouton Abandonner, Recommencer, Ignorer
- « **vbYesNoCancel** » : Affiche le bouton Oui, Non et Annuler
- « **vbRetryCancel** » : Affiche le bouton Recommencer et Annuler
- « **vbCritical** » : Affiche l'icône de message critique et le son associé
- « **vbQuestion** » : Affiche l'icône question et le son associé
- « **vbExclamation** » : Affiche l'icône exclamation et le son associé
- « **vbInformation** » : Affiche l'icône information et le son associé
- « **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
- « **vbSystemModal** » : Toutes les applications sont suspendues jusqu'à ce que l'utilisateur ait répondu

Astuce : Vous pouvez ajouter plusieurs boutons en écrivant le code suivant :

```
MsgBox (« Texte », vbYesNo + vbInformation + etc., « Titre »)
```

Si vous voulez faire apparaître une boîte de dialogue n'oubliez pas de mettre « **Call** » avant « **Msgbox** » comme ceci :

```
Sub test()
        Call MsgBox (« Texte », vbYesNo, « Titre »)
End Sub
```

Si vous ne mettez pas « **Call** », un message d'erreur apparaît dans le cas des « **MsgBox** » faisant appel à divers boutons et modifications. Nous vous conseillons de mettre « **Call** » devant chaque « **MsgBox** » que vous êtes amenés à utiliser afin d'éviter de l'oublier lorsque cela sera nécessaire.

II. LES « INPUTBOX »

La fonction « **InputBox** » fonctionne de la même manière que « **MsgBox** », sauf qu'il n'y a pas de bouton. La fonction « **InputBox** » demande à un utilisateur de rentrer un nombre ou du texte. Elle est très utile si vous voulez interagir avec l'utilisateur.

En voici la syntaxe : « **InputBox (« Texte », « Titre », valeur pas défaut)** ».

La valeur par défaut représente la valeur qui sera inscrite quand l'utilisateur verra l' « **InputBox** ». Par exemple, nous voulons créer une « **InputBox** » dans laquelle l'utilisateur rentrera son prénom :

```
Sub Prénom ()
        VotreNom = Application.Username
        Call InputBox (« Rentrez votre prénom », « renseignements », VotreNom)
End Sub
```

III. LES EVENEMENTS

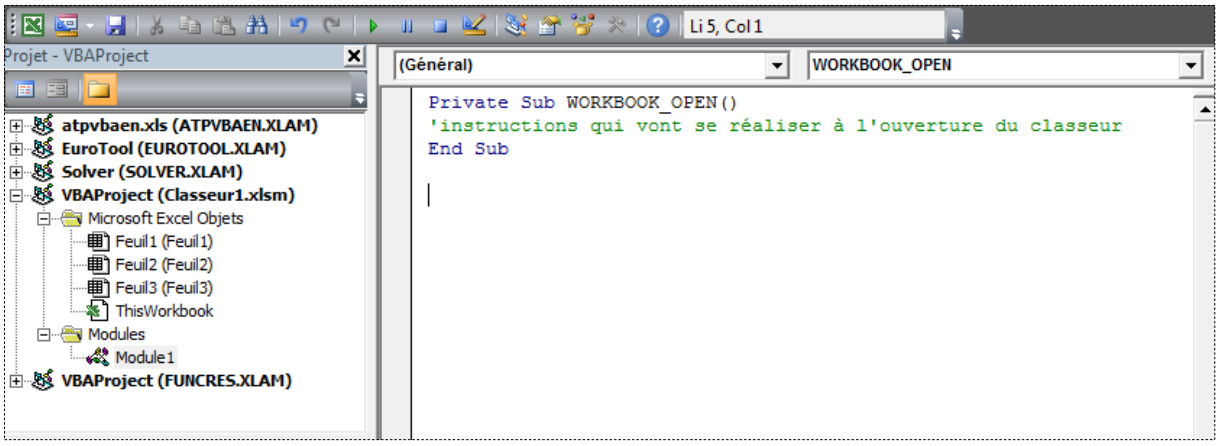
Les évènements WORKBOOK

Il est possible d'écrire une procédure en fonction d'un évènement qui se passe sur le classeur (« **WORKBOOK** »).
Ces procédures sont à écrire dans la feuille « **ThisWorkbook** » dans VBA.

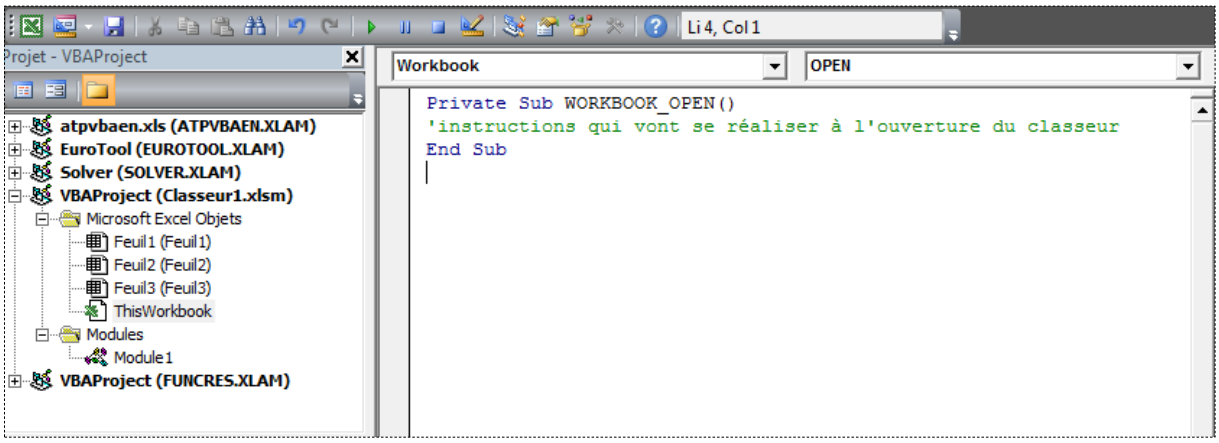
Par exemple, si l'on veut exécuter des instructions à l'ouverture d'un classeur, il faudra placer les instructions dans la procédure qui s'appelle :

```
Private Sub WORKBOOK_OPEN ()
'Instructions qui vont se réaliser à l'ouverture du classeur
End Sub
```

Cette procédure devra être placée dans la feuille « **ThisWorkbook** ». En effet, si on place cette procédure dans un simple module, cela va donner :



Alors que si on l'écrit dans « **ThisWorkbook** » :



Sur ce dernier écran, la chose primordiale qui a changé est le fait que le menu déroulant en haut à gauche a pris la valeur « Workbook » et celui de droite « Open ». A partir de ce moment-là, on peut en déduire que VBA a compris que cette procédure a pour but de lancer les instructions à l'ouverture du classeur.

Par exemple, si je veux afficher un message pour dire bonjour à l'utilisateur à l'ouverture du classeur, dans « **ThisWorkbook** » je vais écrire comme procédure :

```
Private Sub WORKBOOK_OPEN ()
        MsgBox("Bonjour " & Application.Username & " !")
End Sub
```

A noter : pour que les macros se lancent à l'ouverture d'un fichier Excel il faut aller dans l'onglet Développeur puis « Sécurité des macros » dans la première partie tout à gauche. Enfin, il faut cocher : « Activer toutes les macros ». Attention cependant ! Nous ne pouvons que vous conseillez de désactiver cette option si vous ouvrez un fichier dont vous ne connaissez pas la provenance et qui contient des macros. En effet, on peut supprimer le disque dur d'un ordinateur via une simple macro VBA par exemple...

Il existe une panoplie complète de « **Private Sub** » qui, une fois écrites dans l'onglet « **ThisWorkbook** » sous VBA, permettent de lancer des instructions lorsque certains évènements se produisent.

Voici une liste de noms de « **Private Sub** » non exhaustive :

- « **WORKBOOK_OPEN** » : exécute les instructions à l'ouverture du classeur.
- « **WORKBOOK_BEFORECLOSE** » : exécute les instructions juste avant l'ouverture du classeur.
- « **WORKBOOK_BEFORESAVE** » : exécute les instructions juste avant de sauvegarder le fichier.
- « **WORKBOOK_AFTERSAVE** » : exécute les instructions juste après avoir sauvegardé le fichier.
- « **WORKBOOK_BEFOREPRINT** » : exécute les instructions juste avant de lancer l'impression.
- « **WORKBOOK_SHEETACTIVATE** » : exécute les instructions à chaque changement de feuille sur le classeur.
- « **WORKBOOK_SHEETBEFOREDOUBLECLICK** » : exécute les instructions avant un double clic dans une feuille du classeur.
- « **WORKBOOK_SHEETBEFORERIGHTCLICK** » : exécute les instructions avant un clic droit dans la feuille de calcul.
- « **WORKBOOK_SHEETCHANGE** » : exécute les instructions à chaque changement dans l'une des feuilles du classeur.

- « **WORKBOOK_SHEETCALCULATE** » : exécute les instructions quand les calculs sont lancés dans l'une des feuilles du classeur.
- « **WORKBOOKSHEETSELECTIONCHANGE** » : exécute les instructions à chaque changement de sélection.
- « **WORKBOOK_NEWSHEET** » : exécute les instructions quand on crée une nouvelle feuille dans le classeur.
- « **WORKBOOK_SHEETFOLLOWHYPERLINK** » : exécute les instructions quand on clique sur un lien hypertexte.
- « **WORKBOOK_SELECTIONCHANGE** » : exécute les instructions quand on change la sélection.

A noter : toutes ces « **Private Sub** » existent et sont reconnues quand on les écrit dans l'onglet « **ThisWorkbook** » sous VBA. Cependant, il faut noter que parfois, suivant les instructions que l'on demande, il peut y avoir des bugs et ces procédures ne s'exécuteront pas forcément.

A noter : on peut réguler ces évènements et donc l'exécution des macros liée à ces évènements en désactivant temporairement tous les évènements en écrivant le code suivant dans un module par exemple :

```
Sub désactivation()
        Application.EnableEvents = False
        'Instructions
        Application.EnableEvents = True
End Sub
```

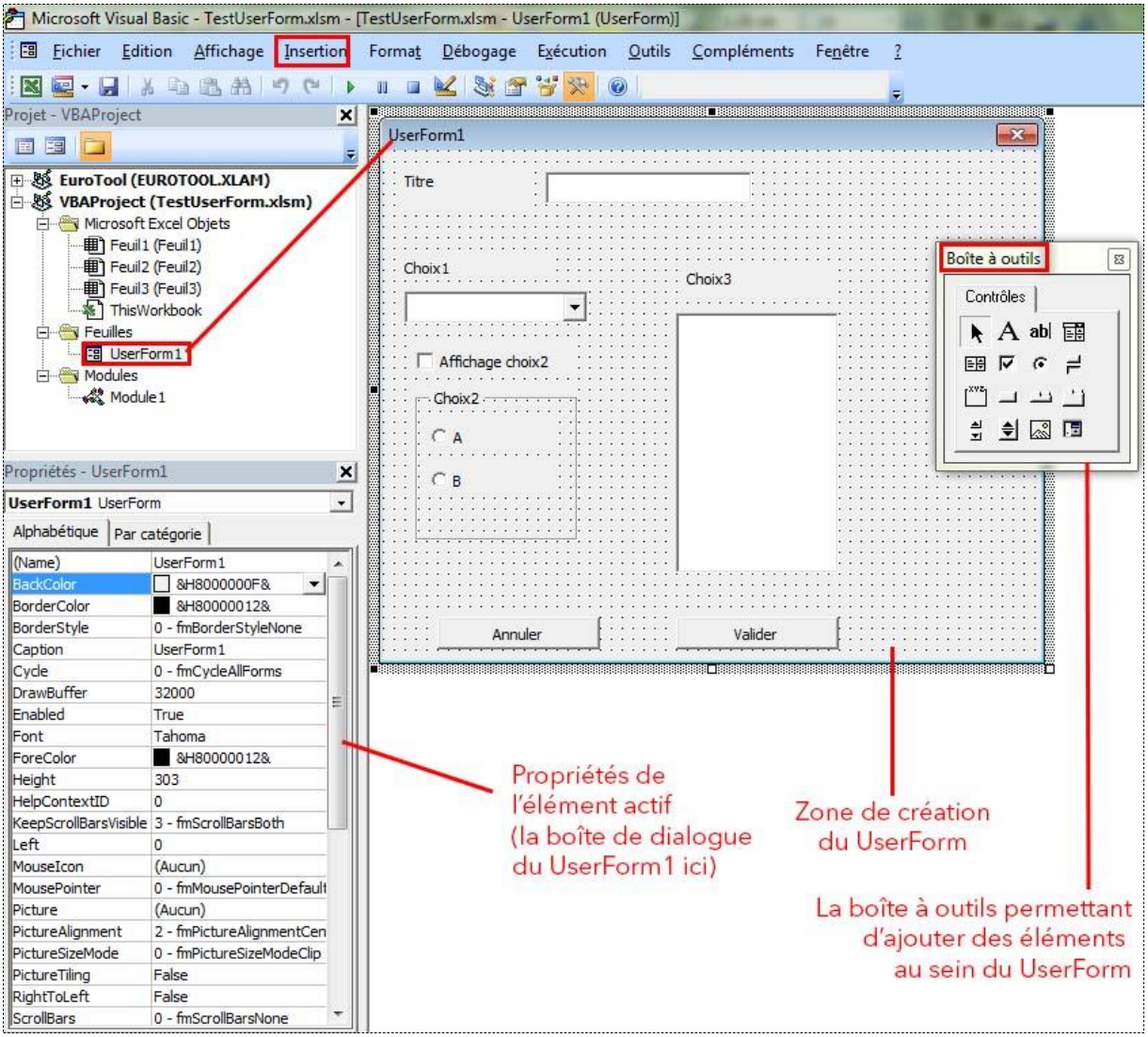
Si vous avez des macros liées à des évènements (ouverture du classeur, sauvegarde, etc.) alors elles ne seront pas exécutées dès lors que la macro « désactivation » va être lancée. Ceci est dû à la présence du « **Application.EnableEvents = False** » au début de la macro. En revanche dès que son exécution sera terminée, les « **Private Sub** » liées aux évènements pourront s'exécuter (à cause du « **Application.EnableEvents = True** ») à la fin de la macro « désactivation ».

IV. LES « USERFORM » (FORMULAIRES)

Un « UserForm », c'est quoi ?

Un formulaire créé pour interagir avec l'utilisateur, tout simplement. Il peut vous demander des valeurs ou encore être utilisé pour tracer des graphiques. Les possibilités sont illimitées. Il permet d'aller plus loin que ce que nous permet un simple « **InputBox** » ou « **Msgbox** » en proposant à l'utilisateur d'avoir un vrai dialogue avec le classeur Excel sur lequel il travaille. Tout comme un « **InputBox** », l'utilisateur pourra entrer des caractères mais il pourra également sélectionner des options. Par exemple, la feuille contient 2 colonnes de nombres, température et pression atmosphérique. Le formulaire peut demander à l'utilisateur sur quelle colonne il souhaite réaliser une moyenne : à partir de là, on peut également imaginer que le formulaire propose à l'utilisateur de réaliser un graphique illustrant les variations de température en fonction de la pression atmosphérique.

Comme on le voit dans cet exemple, on comprend qu'à l'instar des « **InputBox** » et « **MsgBox** », le formulaire peut se nourrir d'informations données par l'utilisateur (ce dernier pourrait nommer le graphique par exemple grâce à une zone texte présente dans le formulaire), mais il peut également se nourrir des informations déjà présentes dans la feuille (dans l'exemple, les données reliées à la température et la pression atmosphérique). Cela multiplie donc les possibilités en termes d'interactions.

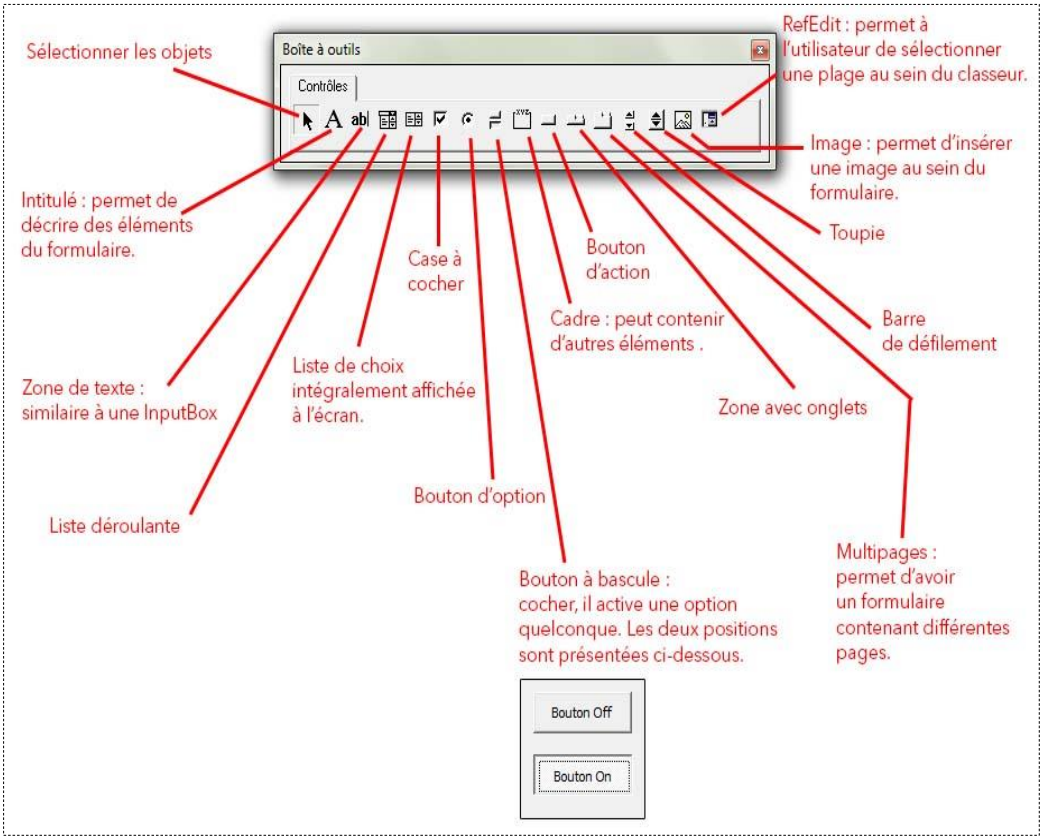


Comment créer un « UserForm » ?

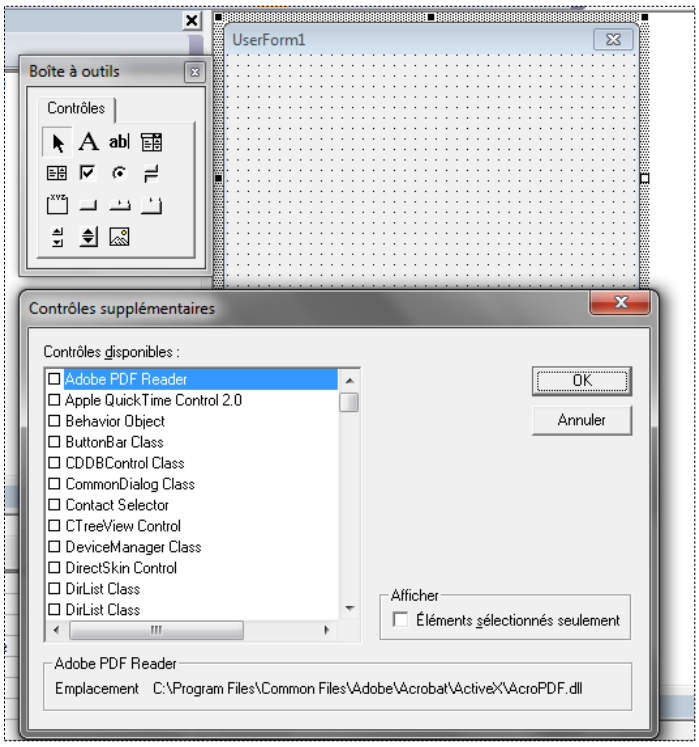
Rien de plus simple une fois de plus ! Comme pour créer un nouveau Module, on se rend dans le menu déroulant Insertion qui se trouve dans l'éditeur Visual Basic et on clique sur « **UserForm** ». Une nouvelle fenêtre apparait alors et vous donne accès entre autres à une boîte à outils permettant de créer différents éléments qui seront insérés dans le formulaire.

Comment ajouter des éléments d'interaction dans un « UserForm » ?

On vient de parler d'une boîte à outils non ? C'est là que va commencer la création d'un « **UserForm** ». On y trouve plusieurs éléments très utiles à la conception de notre formulaire. Passons-les en revue :



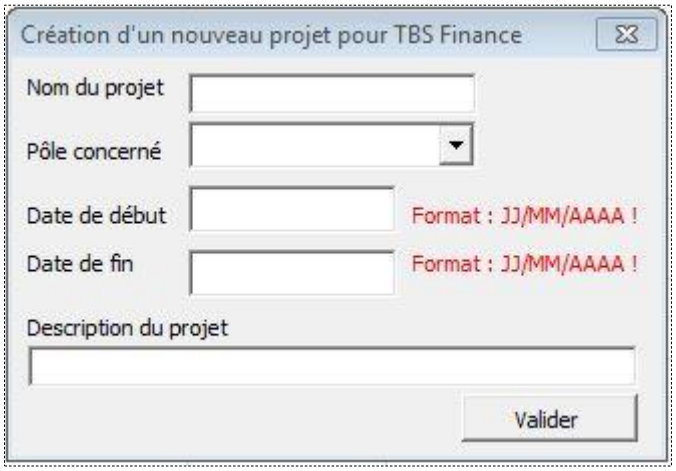
Mais ce ne sont pas les seuls éléments (ou contrôles) disponibles pour créer son formulaire. En effet, en cliquant droit dans la boîte à outils, un menu déroulant s'affiche et l'on peut alors avoir accès à des contrôles supplémentaires en cliquant sur l'intitulé du même nom. On ne détaillera pas tous les contrôles, la liste étant bien trop longue. Mais vous remarquerez par vous-mêmes des noms faisant référence à des logiciels que vous connaissez déjà. VBA sait interagir avec son environnement comme nous l'avons déjà vu avec la fonction « **Shell** » !

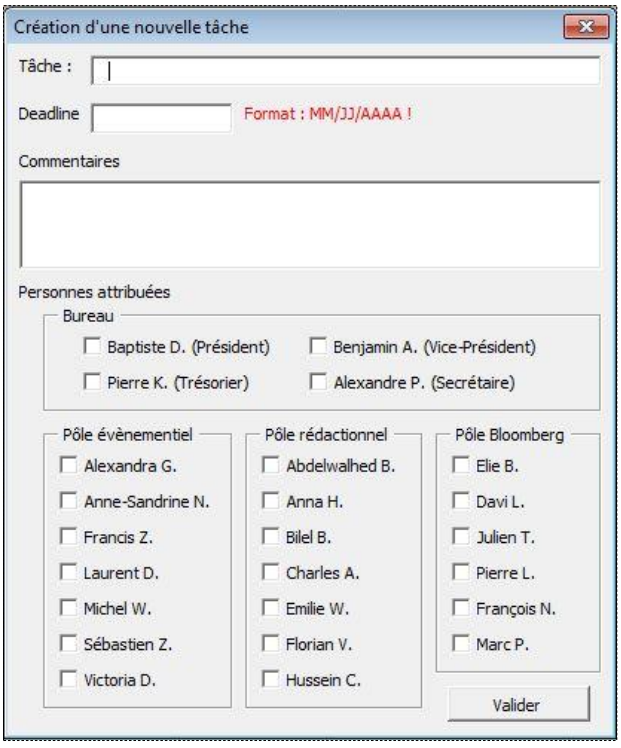

UserForm1
Boîte à outils
Contrôles
Contrôles supplémentaires
Contrôles disponibles :
Adobe PDF Reader
Apple QuickTime Control 2.0
Behavior Object
ButtonBar Class
CDDBControl Class
CommonDialog Class
Contact Selector
CTreeView Control
DeviceManager Class
DirectSkin Control
DirList Class
DirList Class
OK
Annuler
Afficher
Éléments sélectionnés seulement
Adobe PDF Reader
Emplacement C:\Program Files\Common Files\Adobe\Acrobat\ActiveX\AcroPDF.dll

J'ai des nouveaux éléments d'interaction, mais j'aimerais modifier leur couleur, police, forme, taille, etc. comment faire pour avoir un formulaire personnalisé ?

Vous vous souvenez d'un menu présent en bas à gauche dans la fenêtre de l'éditeur Visual Basic ? Ce sont les propriétés de l'élément actif. Pour faire simple, lorsque vous sélectionnez un élément (ou un contrôle) présent au sein de votre formulaire (ou bien votre formulaire lui-même), tous les propriétés de cet élément apparaissent dans cette boîte. L'éditeur vous laisse alors le choix de les classer par ordre alphabétique ou par catégorie. Nous vous conseillons de le faire par catégorie, vous vous y retrouverez plus facilement. Au sein de ce menu, vous pouvez modifier toutes les propriétés d'un élément : il vous suffit de cliquer sur la valeur associée à une propriété et un menu déroulant apparaît vous laissant la possibilité de modifier selon vos préférences la propriété en question.

Voici quelques exemples de formulaires que l'on peut faire avec VBA :



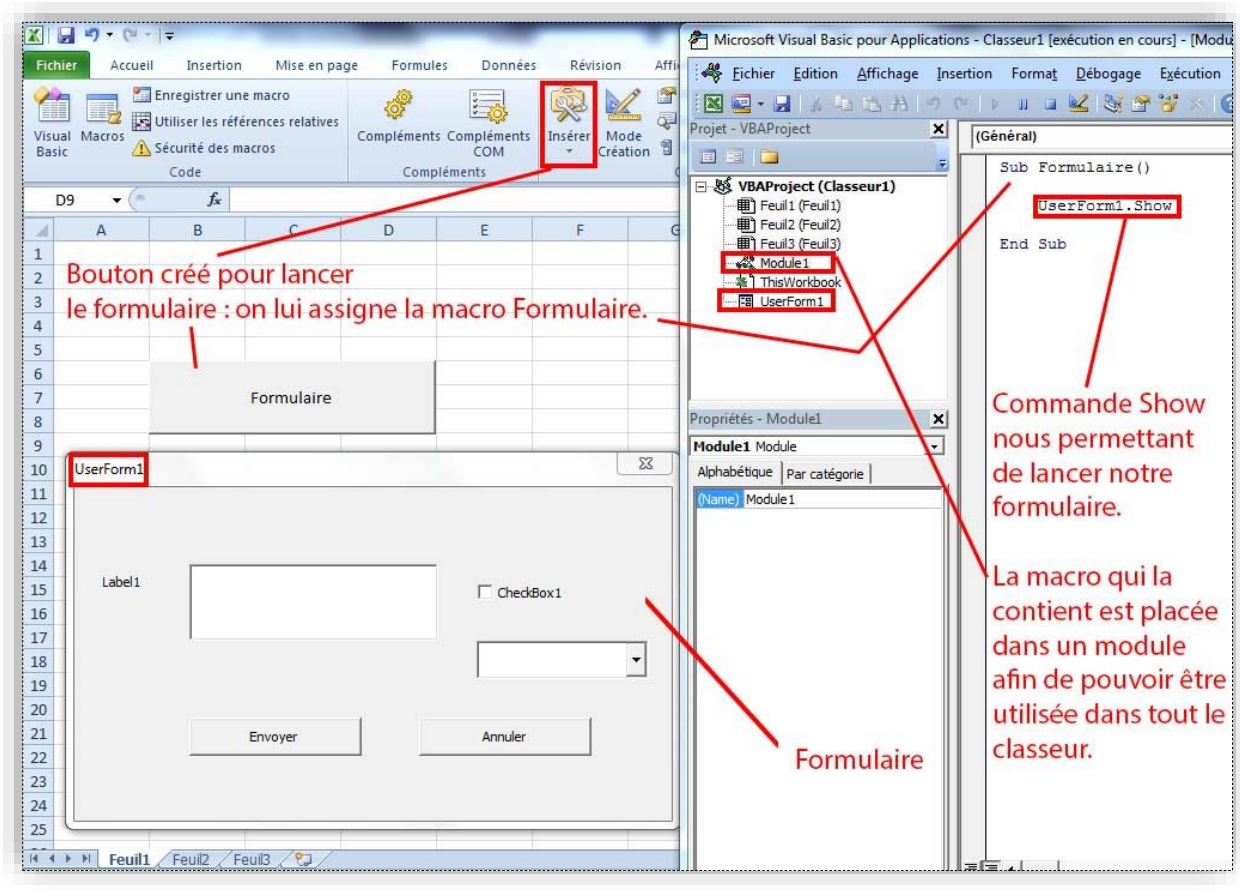


Votre formulaire est mis en forme, tout est parfait, il ne vous reste plus qu'à le soumettre à l'utilisateur (vous en premier lieu). Mais comment faire ?

Alors là, deux possibilités s'offrent à vous en tant que développeur. Exécuter le formulaire en cliquant sur le bouton de lecture vert présent sous la barre des menus. Cette possibilité vous permet d'avoir rapidement accès au résultat ce qui est très utile lorsque l'on code. Problème : l'utilisateur lui, ne va pas aller dans l'onglet Développeur puis dans l'éditeur puis sélectionner votre « **UserForm** » puis appuyer sur ce petit bouton pour enfin avoir accès au formulaire qui lui était dédié. C'est là qu'intervient la deuxième solution.

Utiliser la méthode « Show » pour lancer votre UserForm : comment s'y prendre ?

Une image sera sans doute plus parlante :



L'idée générale est plutôt simple : on va donner la possibilité à l'utilisateur d'afficher le formulaire en cliquant sur un bouton. Comme on l'a déjà vu, il faut donc assigner à ce bouton une macro. La macro en question contient par conséquent un code permettant d'afficher le formulaire. C'est ce qui est affiché sur cette image : l'instruction permettant de lancer le formulaire est donc « **UserForm1.Show** ». Pour rappel, le code se découpe de la manière suivante :

- Tout d'abord, on retrouve l'objet qui va être utilisé : ici, « **UserForm1** », notre formulaire. Vous retrouvez le nom de votre formulaire dans le menu de gauche de l'éditeur Visual Basic. Nous avions fait l'analogie avec une recherche d'adresse : ici, de la même manière il faut donner l'adresse de l'objet qui sera affecté par les actions qui suivront.
- Une fois l'objet spécifié, on place un point pour séparer l'objet de la méthode. Ici on utilise la méthode « **Show** » pour lancer le « **UserForm1** ».

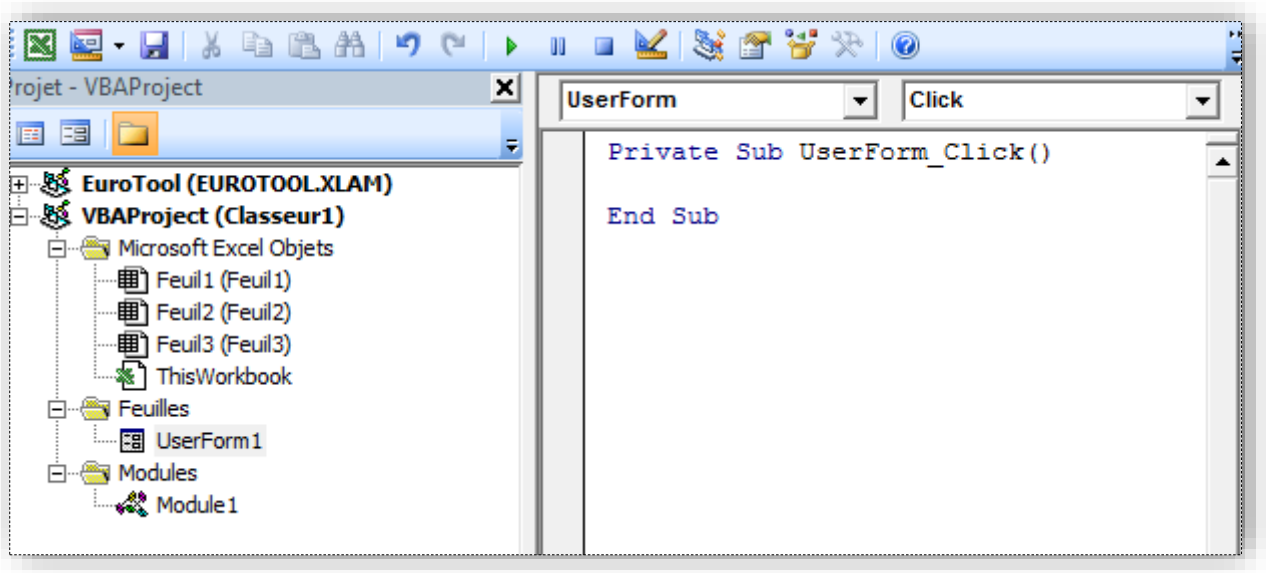
Comme vous pouvez le remarquer, cela marche comme d'habitude : « **Objet.Méthode** ».

Concrètement, on vient de voir comment lancer un formulaire grâce à un bouton. Mais on peut également imaginer que la méthode « **Show** » soit intégrée à une macro « **WORKBOOK_OPEN** ». **Dans ce cas-là, le formulaire s'ouvrira directement à l'ouverture du classeur.** On peut par exemple imaginer un classeur servant à faire du retraitement de données comptables : on rentre dans le formulaire des données du bilan et du compte de résultat puis une fois le formulaire envoyé, le retraitement se fait automatiquement et une feuille est générée.

Pour revenir au formulaire que nous venons d'afficher, vous pouvez remarquer que rien de concret n'est faisable avec : les boutons ne fonctionnent pas, les listes déroulantes sont vides, etc. En l'état, ce formulaire est donc inutile cat il ne récupère pas les informations saisies par l'utilisateur et ne propose aucune option. Pour rendre ce formulaire fonctionnel, nous allons devoir assigner à chaque élément (ou contrôle) qu'il contient une macro selon l'événement qui concernera le contrôle (un clic de souris, l'insertion d'un texte, etc.), tout comme on peut créer un bouton au sein d'une feuille Excel et lui assigner une macro afin de le rendre fonctionnel. Le raisonnement est le même ici.

Rendre un formulaire fonctionnel :

Pour rappel, les contrôles sont des éléments appartenant au formulaire. Les événements qu'ils reçoivent doivent donc être gérés dans le module de code de celui-ci. Le formulaire possède aussi ses propres événements. Commençons par gérer l'initialisation du formulaire, c'est-à-dire l'événement qui sera levé quand le formulaire sera activé. Pour cela, retourner dans votre formulaire et appuyez sur F7 pour afficher le module du code du formulaire. La première chose que vous allez voir apparaitre est alors ceci :



Cette étape peut paraitre sans intérêt, mais en vérité elle apparait très importante quand on travaille sur des formulaires : c'est dans cette partie que l'on va mettre les valeurs par défaut de toutes les variables pouvant entrer en jeu et c'est également là qu'on va définir les propriétés des objets. Il ne faut donc pas omettre ce passage. On peut également dans cette partie du code donner des informations plus esthétiques comme la taille de la fenêtre du formulaire par exemple. Voici un exemple de code :

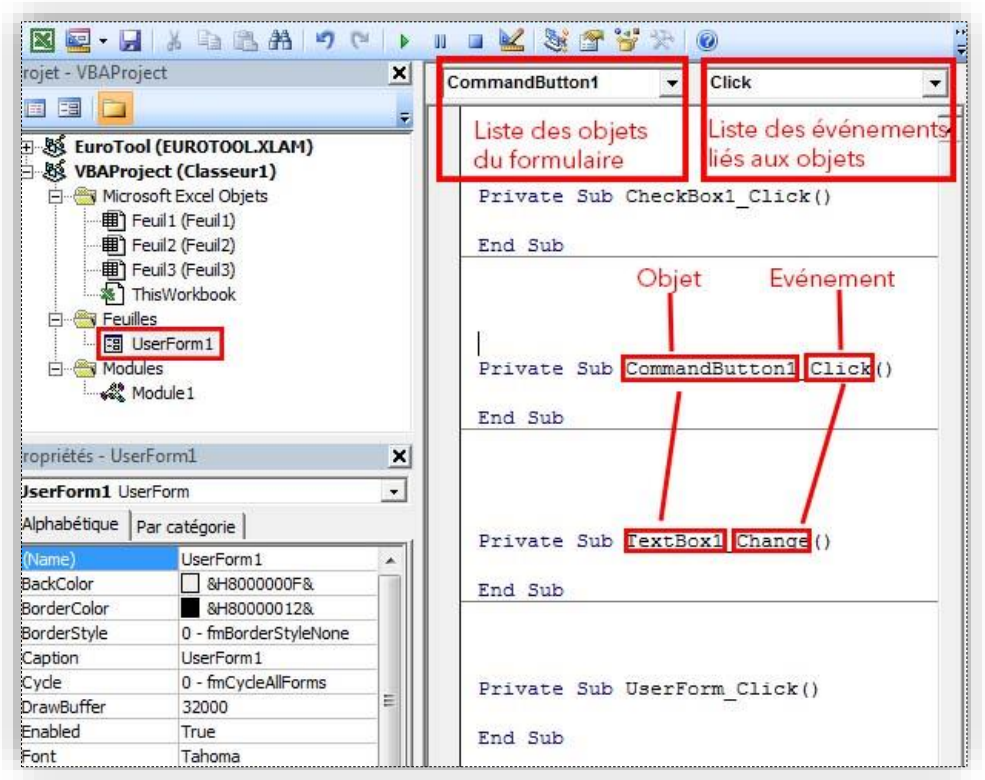
```
Private Sub UserForm_Initialize()
        maVariable = "x"
        CheckBox1 = False
        CheckBox2 = True
        TextBox1 = "mon texte"
        TextBox2 = Range("A1")
End Sub
```

Décryptons ensemble une partie de ce code : « Private Sub UserForm_Initialize ». Comment bien comprendre cette expression ?

Nous avons précédemment expliquer ce qu'était une « **Private Sub** », nul besoin de s'étendre longuement sur ce point. Toutefois, précisons bien une chose : l' « **UserForm** » a ses propres procédures qui sont des « **Private Sub** ». En effet, on ne peut pas les lancer via un bouton sur une feuille de calcul par exemple (elles n'apparaissent pas dans la liste des macros, tout simplement). Ces procédures vont se lancer en fonction des évènements qui sont liés à l' « **UserForm** ». Au sein de ce code, on peut repérer les différents éléments que l'on vient de citer, à savoir Objet et Evénement. Dans le code « **Private Sub UserForm_Initialize** »,

- « **UserForm** » est l'objet.
- « **Initialize** » est l'événement.

Pour traduire le code « **Private Sub UserForm_Initialize** » en français, on pourrait dire : Procédure privée (et donc non visible) d'initialisation du formulaire. Concrètement, lorsque le formulaire (= l'objet) va se lancer (s'initialiser = l'événement), tout le code compris dans cette macro (cette procédure, la « **Private Sub UserForm_Initialize** ») sera interprété par VBA. C'est donc un événement clé. Mais il en existe bien d'autres qui sont associés à divers objets du formulaire. Vous les trouverez dans l'éditeur Visual Basic, après avoir tapé F7 quand vous vous situez sur votre formulaire en construction (cf. image ci-dessous).



Le code VBA est donc très structuré et on rencontre souvent des similarités dans la logique de codage. Par exemple, la manière dont est ordonnée le nom des procédures au sein d'un « **UserForm** » (« **UserForm_Initialize** » par exemple) n'est pas sans rappelé ce que l'on a pu voir dans les cours précédents : Objet puis Méthode, comme par exemple « **Range(« A1 »).Select** ». Ce sont des automatismes à retenir car ils aident à comprendre plus facilement le code.

Pour revenir aux deux menus déroulants présents sur l'image, vous pouvez en les examinant retrouver tous les événements liés à un Objet. La liste est donc très longue, mais certains événements sont des incontournables. Pour ne citer que deux d'entre eux :

- « **UserForm_Initialize** » : déjà présenté.
- « **CommandButton_Click** » : cette procédure définira l'action résultant d'un clic sur un bouton présent dans le formulaire. Pourquoi est-il incontournable ? Tout simplement parce qu'il permet de créer les procédures permettant de Valider le formulaire ou au contraire d'Annuler celui-ci.

La liste est extrêmement longue et beaucoup d'événements peuvent être aisément compris à la simple lecture de leur nom. Si jamais un événement ne vous parait pas clair, nous vous conseillons de faire une recherche sur Internet, vous trouverez très facilement de la documentation et donc une description de l'événement.

Maintenant, souvenons nous de ce que nous souhaitions faire : rendre fonctionnel les contrôles pour avoir un formulaire utilisable. Eh bien, rien de plus simple : il suffit d'insérer de chaque procédure du formulaire reliée à un contrôle des lignes de code définissant les « actions » du contrôle en question. Bien entendu, selon le contrôle, le contenu du code varira. Mais ce qu'il faut retenir c'est que tout ou presque est envisageable. Vous pouvez définir un bouton « Valider » et dans l'éditeur Visual Basic avoir une procédure qui suit schématiquement cette logique par exemple :

```
Private Sub CommandButtonValider_Click ()

'Instructions : Assigner à une variable le contenu de la TextBox1 // Assigner à une variable le contenu d'une plage
'sélectionnée avec RefEdit // Construire un graphique à partir de la plage sélectionnée // …

End Sub
```

Ce typed'instructions va permettre de rendre fonctionnel différents contrôles.

Si l'on récapitule, nous avons vu :

- Comment créer un formulaire.
- Comment y ajouter des contrôles (les éléments d'interaction : les listes déroulantes, les boîtes de texte, etc).
- Comment personnaliser le formulaire.
- Comment le soumettre à l'utilisateur et donc comment utiliser la méthode Show.
- La différence entre un Objet et un Evénement.
- Comment rendre fonctionnel les contrôles du formulaire.

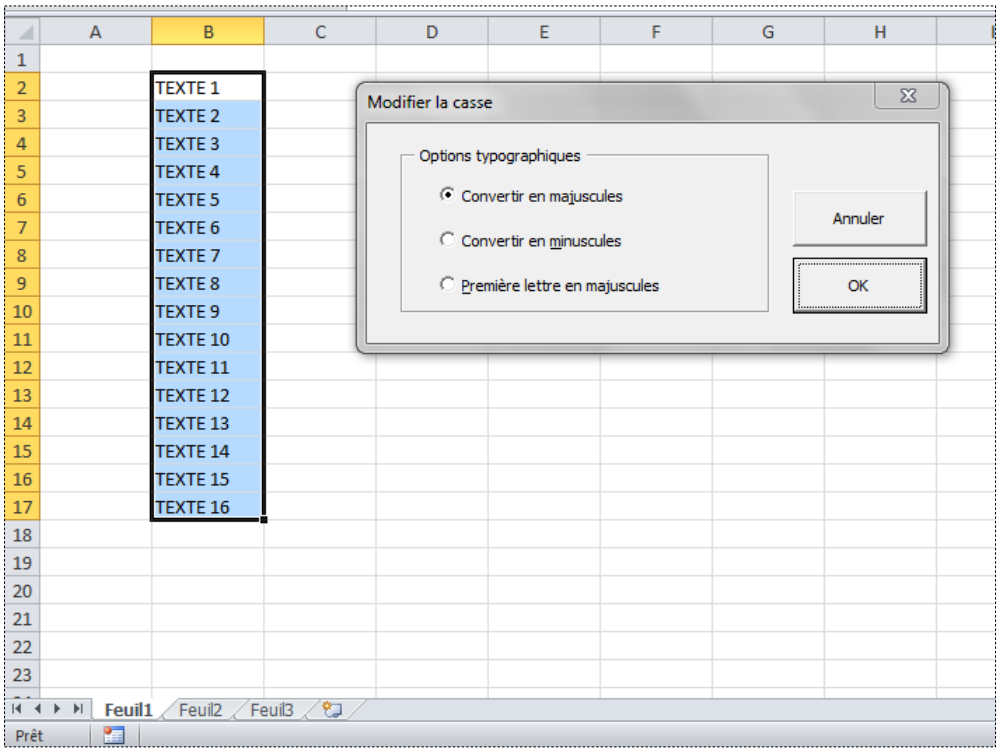
Nous allons désormais faire un exemple qui va nous permettre de bien assimiler le cours.

V. EXEMPLE D'APPLICATION

Objectif : Nous voulons créer une boîte de dialogue qui nous permettra de mettre en majuscules, minuscules ou mettre la première lettre en majuscule le texte d'une plage sélectionnée. Cet exemple nous permettra de revoir également les notions suivantes :

- Condition « **If Then** ».
- Boucle « **For Each In** ».
- « **TypeName** »
- « **Private Sub** » / « **Sub** ».

Voici ce que nous voulons créer :



A vous de jouer !

A. ETAPE 1

1. CREER UN USERFORM

Changer les propriétés suivantes

Caption = Modifier la casse

2. CREER DEUX BOUTONS (OK ET ANNULER) :

Insérer deux boutons de commande. Ensuite modifier les propriétés suivantes pour chaque bouton de commande :

CommandButton1 **Name** = Bouton_OK **Caption** = OK **Default** = True

CommandButton2 **Name** = Bouton_Annuler **Caption** = Annuler **Default** = True

3. CREER UN CADRE

Insérer un cadre et changer les propriétés suivantes :

Name = Cadre **Caption** = Options Typographiques

4. CREER TROIS BOUTONS OPTIONS

Insérer trois boutons options et changer les propriétés suivantes :

Bouton1 **Name** = Option_majuscules **Caption** = Convertir en majuscules **Accelerator** = J **Value** = true

Bouton2 **Name** = Option_Minuscules **Caption** = Convertir en minuscules **Accelerator** = M

Bouton3
Name = Option_1ereCapitale
Caption = Première lettre en majuscule
Accelerator = P

B. TAPE 2

1. DOUBLE CLIQUER SUR LE BOUTON ANNULER

Insérer ce code dans la procédure :

```
Unload Userform1
```

En appuyant sur Annuler, « **Userform1** » disparaît de l'écran

2. CLIQUER SUR LE BOUTON OK

Entrer le code suivant dans la procédure :

```
Private Sub bouton_ok_Click()

Dim Cellule As Range

'Majuscules
If Option_Majuscule Then
For Each Cellule In Selection
Cellule.Value = UCase(Cellule.Value)
Next Cellule
End If

' Minuscules
If Option_minuscules Then
For Each Cellule In Selection
Cellule.Value = LCase(Cellule.Value)
Next Cellule
End If

' Minuscules
If Option_1erecapitale Then
For Each Cellule In Selection
Cellule.Value = Application.WorksheetFunction.Proper (Cellule.Value)
Next Cellule
End If

'Ferme boîte de dialogue
Unload UserForm1
```

```
End Sub
```

Par exemple, si l'utilisateur clique sur le bouton Convertir en majuscule, VBA prend chaque cellule et transforme le texte en majuscule avec la fonction « **UCase** ».

Remarque : « **UCase** » permet de transformer du texte en majuscule, « **LCase** » permet de transformer du texte en minuscule, et « **Application.WorksheetFunction.Proper** » permet de mettre la première lettre d'un texte en majuscule. Toutes ces fonctions sont déjà prédéfinies dans VBA.

C. ETAPE 3

Nous allons enfin afficher la boîte de dialogue :

1) Créer un nouveau module
2) Entrer le code suivant :

```
Sub modifcasse()

'Quitte si une plage n'est pas sélectionnée
If TypeName(Selection) = "Range" Then
'Affichage de la boîte de dialogue
UserForm1.Show

End If
End Sub
```

Astuce : afin de ne pas être obligé de lancer la macro à chaque utilisation, affectez lui un raccourci (par exemple Ctrl + Shift + C).

COURS 5 : LES TABLEAUX

I. PRESENTATION DES TABLEAUX ET DE LEUR INTERET

Les tableaux permettent de stocker des valeurs, de la même manière que vous pouvez le faire sous Excel (sauf qu'il est beaucoup plus rapide de parcourir un tableau VBA qu'un tableau d'une feuille Excel). Ils fonctionnent de la même façon qu'une variable. Il faut d'ailleurs les déclarer dans un premier temps.

Intérêt majeur : **créer un grand nombre de variables en peu de temps, en évitant l'énumération, et cela permet de créer des codes qui s'exécutent plus rapidement quand on traite un grand nombre de variables.**

II. DECLARATION D'UN TABLEAU

A. DECLARATION SIMPLE D'UN TABLEAU

Pour déclarer un tableau à une entrée (= 1 colonne), on écrira :

```
Dim nom_de_mon_tableau(X)
```

X représente le nombre de lignes.

A noter : la numérotation des tableaux commence à 0. Par exemple : « **nom_de_mon_tableau(4)** » est un tableau à 5 lignes (0, 1, 2, 3, 4) et 1 colonne. Ce tableau comporte donc 5 cases.

A noter : il est possible de faire commencer la numérotation des tableaux à 1. Cela est en effet plus pratique**. Il faut alors écrire « Option Base 1 » au tout début de votre module, avant même le début de vos procédures pour que les tableaux commencent à 1 et non 0 (valeur par défaut).** De la même manière, on peut également forcer les tableaux à commencer à 0 en écrivant « **Option Base 0** » au début du module.

B. DECLARATION D'UN TABLEAU A PLUSIEURS ENTREES

Pour déclarer un tableau à plusieurs entrées (= plusieurs colonnes), on écrira :

```
Dim nom_de_mon_tableau(X,Y)
```

X représente le nombre de lignes et Y le nombre de colonnes. La numérotation commence toujours à 0. Par exemple, « **nom_de_mon_tableau(2,3)** » est un tableau à 3 lignes (0, 1, 2) et 4 colonnes (0, 1, 2, 3), ce qui en fait un tableau à 12 cases.

A noter : nous ne sommes pas obligés de déclarer le nombre de lignes ou de colonnes de notre tableau. Pour le déclarer on écrira alors : « **Dim nom_de_mon_tableau()** ». On déclare alors ainsi un tableau dynamique.

C. DECLARER LE TYPE D'INFORMATIONS CONTENUES DANS LES TABLEAUX

De la même manière que pour les variables, on peut déclarer le type d'informations contenues dans un tableau. Par exemple, je peux ne vouloir que des nombres entiers dans mon tableau, je vais alors coder :

```
Sub déclaration()
        Dim mon_tableau(1,9) As Integer
End Sub
```

D. SYNTAXE X TO Y POUR DECLARER UN TABLEAU

Enfin, il existe une syntaxe qui est très utile pour déclarer des tableaux puisqu'elle permet de donner la numérotation de départ et de fin du tableau. Ainsi, on peut se passer de « **Option Base 0** » ou « **Option Base 1** ».

Pour les tableaux non dynamiques, cette déclaration prend la syntaxe suivante :

```
Dim mon_tableau(1 TO 10, 1 TO 4)
```

Au lieu de:

```
Dim mon_tableau(10,4)
'avec Option Base 1 en début de module
```

Pour les tableaux dynamiques :

```
Dim mon_tableau()
Redim mon_tableau(1 To nombre_de_lignes, 1 To nombre_de_colonnes)
```

Au lieu de :

```
Dim mon_tableau()
Redim mon_tableau(nombre_de_lignes, nombre_de_colonnes)
'avec Option Base 1 en début de module.
```

A noter : grâce à cette nouvelle syntaxe dans la déclaration des tableaux, on peut faire commencer les tableaux à 5 ou même 10. Par exemple :

```
Dim mon_tableau(5 To 10, 3 To 5)
```

III. LES FONCTIONS « UBOUND » ET « LBOUND »

La fonction « **UBound** » est très intéressante puisqu'elle permet de connaître le plus grand numéro de ligne ou de colonne de mon tableau.

La fonction « UBound » prend la forme suivante:

```
UBound (nom_tableau, n°_dimension)
```

De même, il existe la fonction « **LBound** » qui permet de connaître le plus petit numéro de ligne ou de colonne de mon tableau.

La fonction « LBound » prend la forme suivante :

```
LBound (nom_tableau, n°_dimension)
```

A noter : n°_dimension (numéro de dimension) vaut 1 si vous vous intéressez aux lignes, 2 si vous vous intéressez aux colonnes.

Ainsi, si je veux connaître le nombre de lignes de mon_tableau je vais coder :

```
nombre_de_lignes = UBound (mon_tableau, 1) – LBound (mon_tableau, 1) + 1
```

A noter : le « n°_dimension » (numéro de dimension) n'est pas obligatoire, par défaut c'est la première dimension que « **UBound** » prendra en compte si celle-ci n'est pas précisée.

De même, si je veux connaître le nombre de colonnes de mon_tableau, je vais coder :

```
nombre_de_colonnes = UBound (mon_tableau, 2) – LBound (mon_tableau, 2) + 1
```

IV. ENREGISTREMENT DES DONNEES D'UN TABLEAU

A. ENREGISTREMENT SIMPLE

Pour enregistrer les données dans une case d'un tableau il suffit d'utiliser chaque case du tableau comme une variable. Supposons que j'ai un tableau de 10 cases (1 colonne) et que je veuille mettre le contenu de la case A1 dans la première case du tableau je vais alors coder :

```
Sub enregistrementsimple()
        Dim mon_tableau(9)
        'mon_tableau aura alors bien 10 cases
        mon_tableau(0) = Range("A1")
        MsgBox mon_tableau(0)
End Sub
```

Par conséquent, pour enregistrer le contenu des cellules A1 à A10 dans un tableau on écrira :

```
Sub enregistrement()
        Dim mon_tableau(9)
        For i = 0 To 9
                mon_tableau(i) = Range("A"& i+1)
        Next

        For i = LBound(mon_tableau) To UBound(mon_tableau)
                MsgBox mon_tableau(i)
        Next
End Sub
```

B. ENREGISTREMENT D'UNE PLAGE DE CELLULES

Si l'on veut enregistrer dans un tableau le contenu des cellules A1 à H10, on écrira :

```
Sub enregistrement_plage()
        Dim mon_tableau(9,7)
        For i = 0 To 9
                For j = 0 To 8
                        mon_tableau(i,j)=cells(i+1,j+1)
                Next
        Next
End Sub
```

A noter : on peut enregistrer toute une plage de cellules dans un tableau sans boucle grâce à la ligne de code suivante par exemple : « **mon_tableau = Range (« A1:F56 »).Value** ».

VBA va alors comprendre qu'il faut mettre dans la première case du tableau la valeur de la case A1, etc. Attention cependant, le premier numéro du tableau sera alors 1 et non 0 comme vu précédemment, cela peut donc être source de confusion. Exemple : la valeur de la case A1 sera dans mon_tableau(1,1) et non dans mon_tableau(0,0).

V. LES TABLEAUX DYNAMIQUES ET LA FONCTION « REDIM »

Parfois, on peut avoir envie de faire un tableau dynamique, i.e. un tableau dont le nombre de lignes ou de colonnes change en fonction de modifications que l'on peut être amené à faire.

Le problème ici est que l'on ne peut pas mettre le numéro de lignes ou de colonnes (puisque, par défaut, il est censé varier) dans la première déclaration d'un tableau.

En effet, pour contourner ce problème, il suffit de déclarer d'abord votre tableau de manière classique et d'utiliser ensuite la fonction **« ReDim »** pour déclarer une nouvelle fois votre tableau, l'avantage de la fonction « **ReDim** » est que l'on peut utiliser des variables dans la déclaration. Par exemple :

```
Sub déclaration()
        'première déclaration classique :
        Dim mon_tableau()
        'deuxième déclaration pour mettre des variables :
        ReDim mon_tableau(variable1, variable2)
End Sub
```

« **variable1** » et « **variable2** » sont deux variables qui peuvent donc prendre des valeurs différentes.

Attention : quand on utilise « ReDim », tout le contenu du tableau est effacé, il y a donc perte de données. Pour éviter ce problème, il faut utiliser la fonction « ReDim Preserve » à la place de « ReDim ».

Rappel : pour connaître le numéro de ligne de la dernière cellule remplie d'un bloc de cellules, on utilise la fonction « **End(xldown)** ». En l'occurrence, si on veut le numéro de ligne de la dernière cellule remplie de la colonne A1 on codera:

```
Sub numéroligne()
        numéro_ligne = Range("A1").End(xlDown).Row
End Sub
```

VI. LA FONCTION « ARRAY »

La fonction « **Array** » permet de remplir un tableau de manière plus rapide car on peut mettre les valeurs d'un tableau à une dimension les unes à la suite des autres. Par exemple, au lieu d'écrire :

```
Sub tableau()
        Dim mon_tableau(5)
        mon_tableau(0) = "Bonjour"
        mon_tableau(1) = "la"
        mon_tableau(2) = "majeure"
        mon_tableau(3) = "finance"
        mon_tableau(4) = "de"
        mon_tableau(5) = "TBS"
End Sub
```

On va plutôt écrire :

```
Sub tableau()
        Dim mon_tableau() As Variant
        'Par défaut, lorsqu'on ne le précise pas, un tableau contient des données de type « Variant », le « As Variant » n'est donc pas obligatoire ici.
        mon_tableau = Array("Bonjour", "la", "majeure", "finance", "de", "TBS")
End Sub
```

Cette deuxième solution est beaucoup plus rapide et moins fastidieuse à écrire.

A noter : **quand on utilise « Array », on ne doit pas déclarer un tableau fixe (qui par essence est incompatible avec la fonction « Array »**, cette dernière pouvant potentiellement apporter plus de variables que ce que le tableau ne peut recevoir) mais un tableau dynamique.

A noter : il est indispensable d'utiliser le type « Variant » quand on déclare une variable tableau qui fonctionnera avec « Array ». L'autre solution est de ne pas déclarer le type de la variable. Mais si vous voulez déclarer le type de données contenues dans votre tableau, seul le type « **Variant** » fonctionne.

VII. UNE AUTRE FONCTION UTILE : « REPLACE »

Cette fonction permet de remplacer toute une série de valeurs par une autre série indiquée par l'utilisateur.

Prenons l'exemple suivant qui nous vient du site http://silkyroad.developpez.com :

```
'Remplace les points virgules par des points dans la plage de cellules sélectionnée
Sub remplacerCaracteres()
        Dim Cellule As Variant
        For Each Cellule In Selection
        Cellule.Value = Replace(Cellule.Value, ";", ".")
         Next Cellule
End Sub
```

Retournez à présent sur votre feuille Excel : créez une plage de cellules qui contient des « **;** » puis créez un bouton auquel vous allez assigner cette procédure. Une fois tout cela fait, sélectionnez la plage de cellules que vous venez de créer et appuyez sur le bouton. Observez le résultat : les « **;** » ont été remplacés par des « **.** ».

Cela nous permet donc de comprendre la syntaxe de la fonction « **Replace** » :

```
Replace ( chaine_à_traiter, sous-chaine_à_remplacer , sous-chaine_de_remplacement)
```

Et cela peut être utilisé avec des tableaux : les sous-chaines à remplacer et les sous-chaines de remplacement peuvent très bien être des variables tableaux. Par exemple, on peut remplir ces deux variables de la manière suivante :

```
sous-chaine_à_remplacer = Array (« A », « B », « C »)
sous-chaine_de_remplacement = Array (« 1 », « 2 », « 3 »)

For i = LBound(sous-chaine_à_remplacer) to UBound (sous-chaine_à_remplacer)
        chaine_à_traiter = Replace ( chaine_à_traiter, sous-chaine_à_remplacer(i) , sous-chaine_de_remplacement(i))
Next i
```

Vous devrez donc spécifier à quoi correspond la « **chaine_à_traiter** » et le remplacement se fera. Cependant, il faut être prudent quant à l'utilisation de la fonction « **Replace** » : en effet, elle va traiter les chaines de caractères de manière bien particulière. Imaginons que l'on veuille créer une traduction du français vers l'anglais, on va avoir envie de procéder ainsi :

```
Sub tableau()

        ma_chaine_de_caracteres = "Bonjour aux étudiants de Toulouse"
        MsgBox (ma_chaine_de_caracteres)        'Affiche la chaine avant modification

        Dim mon_tableau_francais() As Variant    'As Variant est facultatif tout comme la déclaration de la variable
        Dim mon_tableau_anglais() As Variant     'As Variant est facultatif tout comme la déclaration de la variable
        mon_tableau_francais = Array("Bonjour", "aux", "étudiants", "de", "Toulouse")
        mon_tableau_anglais = Array("Hello", "to", "students", "from", "Toulouse")


        For i = 0 To 4      'A noter : nous aurions pu utiliser LBound et UBound comme dans l'exemple précédent
 ma_chaine_de_caracteres = Replace(ma_chaine_de_caracteres, mon_tableau_francais(i), mon_tableau_anglais(i))
        Next i

        MsgBox (ma_chaine_de_caracteres)        'Affiche la chaine modifiée
End Sub
```

On peut donc s'attendre à obtenir avec la dernière « **MsgBox** » le message suivant : « **Hello to students from Toulouse** ». Mais pourtant, nous n'obtenons pas cela mais : « **Hello to stufromnts from Toulouse** ». Comment expliquer cela ? Quand VBA va passer en revue les éléments de notre chaine de caractères, il ne va pas les traiter par blocs. En d'autres termes, quand VBA lit « **Bonjour** », il va lire lettre par lettre. Et cela explique notre « **stufromnts** ». Voilà les étapes qui nous permettent d'afficher un tel résultat :

- Au troisième passage dans la boucle, VBA cherche le terme « **étudiants** » dans la chaine pour le remplacer par « **students** ». Il le localise et fait le remplacement. Notre chaine à ce stade des opérations vaut : « **Hello to students de Toulouse** ».
- Au quatrième passage dans la boucle, VBA cherche le terme « **de** » pour le remplacer par « **from** ». Et comme VBA analyse la chaine en passant lettre par lettre, il va trouver deux fois cette référence puisque le terme « **students** » comprend un « **de** ». Il effectue donc deux remplacements. Notre chaine vaut alors : « **Hello to stufromnts from Toulouse** ».
- Le dernier passage ne modifie rien puisque « **Toulouse** » reste « **Toulouse** ».

Et voilà, c'est aussi simple que ça ! Comme vous le constatez, il faut parfois aborder VBA avec une logique bien particulière pour analyser son fonctionnement.

VIII. LA FONCTION « SPLIT »

La fonction « **Split** » est très utile puisqu'elle permet de séparer une chaîne de caractères pour remplir les cases d'un tableau.

Elle prend la structure suivante : « **Split(ma_chaine_de_caracteres, "séparateur")** »

Par exemple, supposons que l'on ait « **ma_chaine_de_caracteres = "Bonjour,aux,étudiants,de,Toulouse"** » et que je veuille mettre "**Bonjour**" dans la première case de mon tableau, "**aux**" dans la deuxième, etc. alors je vais coder:

```
Sub test_split()
        ma_chaine_de_caracteres = "Bonjour,aux,étudiants,de,Toulouse"
        mon_tableau = Split(ma_chaine_de_caracteres, ",")
End Sub
```

Mais là où la fonction devient plus intéressante c'est que l'espace peut être considéré comme un séparateur. Par exemple:

```
Sub test_split()
        ma_chaine_de_caracteres = "Bonjour aux étudiants de Toulouse"
        mon_tableau = Split(ma_chaine_de_caracteres, " ")
End Sub
```

Cette dernière procédure mettra alors "**Bonjour**" dans la première case de « **mon_tableau** », "**aux**" dans la deuxième case, etc.

Rappel : la procédure suivante renvoie exactement les mêmes valeurs dans le tableau que les deux procédures précédentes :

```
Sub test_split()
        mon_tableau = Array ("Bonjour", "aux", "étudiants", "de", "Toulouse")
End Sub
```

A noter : prenons l'exemple suivant :

```
Sub testsplit()
        Msgbox Split("Bonjour aux étudiants de Toulouse", " ")(3)
End Sub
```

Cette procédure va renvoyer "de". En effet, il faut faire attention au décalage !

IX. LA FONCTION « JOIN »

Il existe la fonction inverse de « **Split** » qui est la fonction « **Join** ». Comme son nom l'indique, cette fonction permet de joindre plusieurs chaînes de caractères pour n'en faire qu'une seule.

Cette fonction a la structure suivante :

« Join (Colonne de tableau, séparateur entre les différents contenus des cases d'une colonne) »

Par exemple :

```
Sub test_join()
        Dim mon_tableau(4)
        mon_tableau(0) = "Bonjour"
        mon_tableau(1) = "aux"
        mon_tableau(2) = "étudiants"
        mon_tableau(3) = "de"
        mon_tableau(4) = "Toulouse"
        joinchaine = Join(mon_tableau, " ")
        Msgbox (joinchaine)
End Sub
```

A noter : on aurait également pu gagner en efficacité grâce à la procédure suivante :

```
Sub test_join()
        ma_chaine_de_caracteres = "Bonjour aux étudiants de Toulouse"
        mon_tableau = Split(ma_chaine_de_caracteres, " ")
        joinchaine = Join(mon_tableau, " ")
        Msgbox (joinchaine)
End Sub
```

A noter : à la place d'une colonne d'un tableau, on peut également utiliser la fonction Array. Par exemple :

```
Sub test_join2()
        ma_chaine = Join( Array("Bonjour", "aux", "étudiants", "de", "Toulouse"), " ")
        MsgBox(ma_chaine)
End Sub
```

ANNEXES

I. EXEMPLE D'APPLICATION : CREER LA FONCTION FACTORIELLE DE DEUX FAÇONS DIFFERENTES

A. AVEC LA BOUCLE « FOR »

```
Function factorielle_for(n As Integer)

        factorielle_for = 1

        For i = 1 To n
                factorielle_for = factorielle_for * i
        Next

End Function
```

B. AVEC UNE FONCTION RÉCURSIVE

```
Function factorielle_recursive(n As Integer)

        If n <= 1 Then
                factorielle_recursive = 1
        Else

                factorielle_recursive = n * factorielle_recursive(n - 1)
        End If

End Function
```

II. LES SITES QUE NOUS VOUS RECOMMANDONS

- http://excel.developpez.com/cours/?page=prog#autres : site d'une grande qualité proposant de nombreux cours très complets et rédigés avec une grande clarté.
- http://silkyroad.developpez.com/ : espace personnel d'un contributeur du site excel.developpez.com qui regroupe de nombreuses ressources.
- http://www.excel-downloads.com/
- http://www.excel-pratique.com/fr/index.php : site proposant une formation à VBA très instructive.
- http://www.blog-excel.com/ : blog très intéressant proposant de nombreuses macros. Ce blog est tenu par le créateur du site excel-pratique.com.
- http://msdn.microsoft.com/fr-fr/office/ff688774.aspx : site de Microsoft qui contient de très nombreuses informations utiles.
- http://optionexplicitvba.blogspot.fr/ : blog très fourni.

Et n'oubliez pas l'aide disponible sous Excel, elle regorge d'informations très utiles ! Pour finir, le site http://projecteuler.net/problems est également très utile : il propose divers problèmes mathématiques que l'on peut

résoudre avec l'outil informatique. C'est une bonne manière de pratiquer et aussi des heures de casse-têtes pour les plus geeks d'entre vous !

REMERCIEMENTS

Nous tenions à remercier tous ceux qui ont pu participer à nos cours et tous ceux qui ont participé à l'élaboration de ceux-ci. Nous remercions également chaleureusement toutes les personnes ayant témoigné un intérêt tout particulier pour cette formation et nous espérons très sincèrement qu'elle vous fournira un atout supplémentaire à faire valoir lors de vos entretiens !

L'équipe TBS Finance.

JIM
JUNIOR INVESTMENT MANAGEMENT by *TBS*