- # Les bases du langage VBA

-

Premiers pas en VBA

Dans cette partie, nous allons approfondir les macros : le temps est venu pour vous de personnaliser vos propres bestioles !

Du VBA, pour quoi faire ?

Relisez le chapitre sur les macros. Nous y avons vu qu'une macro est une série d'instructions. Lorsque vous exécutez cette macro, vous exécutez cette série d'instructions.

Cette fameuse série, elle est écrite quelque part dans un code informatique : le VBA, qui signifie Visual Basic pour Application.

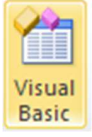
Le VBA a donc besoin d'une application, en l'occurrence ici Excel, pour fonctionner.

Nous avons vu également que pour accéder à ce code, il fallait appuyer sur `Alt` + `F11` mais... c'est un peu le bazar lorsque nous arrivons sur la fenêtre.
Mais alors, à quoi ça sert, le VBA ?
À coder vos propres macros, pardi ! Il y a en effet des macros que vous ne pourrez jamais faire en faisant travailler l'enregistreur, comme compléter la dernière ligne d'un tableau.

Ça sera à vous de les réaliser ! 

L'interface de développement

L'interface de développement, c'est la fenêtre sur laquelle vous tombez lorsque vous appuyez sur `Alt` + `F11` ou encore lorsque vous vous rendez dans l'onglet « *Développeur* », dans le groupe « *Code* » et que vous cliquez sur le bouton :



Si vous ne savez plus comment activer l'onglet « *Développeur* », je vous redirige vers le chapitre précédent.
Nous allons tout d'abord voir comment s'organise un projet et inévitablement, comment fonctionne l'interface de développement.

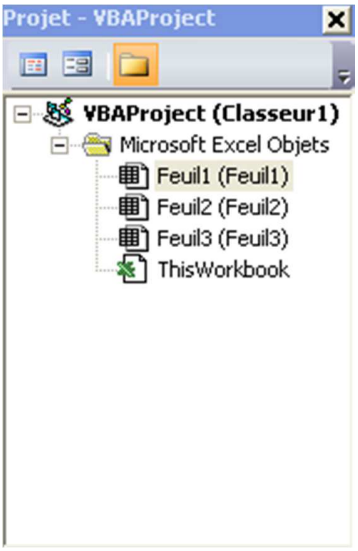
Un projet, oui mais lequel ?

Là, c'est vous qui décidez. Un projet s'applique en général sur un travail dans le classeur ou dans une feuille de calcul particulière. C'est un groupe de macros, qui s'appellent entre elles, qui échangent avec l'utilisateur...

Vous codez une macro dans ce qu'on appelle **un module**. C'est comme une feuille blanche dans laquelle vous allez écrire votre ou vos macros.

Bien évidemment, vous pouvez rajouter à votre projet autant de modules que vous voulez. C'est-à-dire que vous pouvez écrire une macro par module, si vous le souhaitez.

Ouvrez MVBA et regardez le menu de gauche :

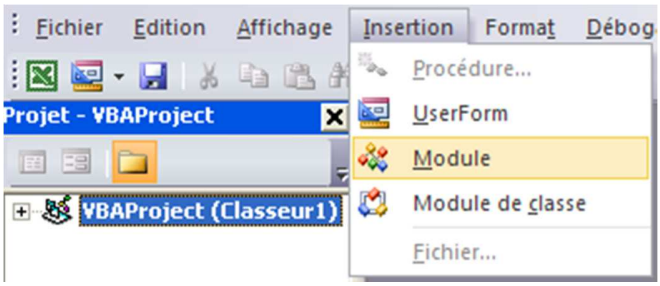


Chaque mot en gras est un projet. Vous pouvez l'explorer au moyen de la petite croix à gauche de chacun :

Ici, je n'ai en fait qu'un seul projet, les autres étant propres à l'application et protégés par mot de passe.

Vous avez tous le même projet : « *VBAProject (Classeur1)* ». Si votre classeur a pour nom *Salariés*, votre projet a pour nom « *VBAProject (Salariés)* ».

Nous allons ajouter un nouveau module qui va vous permettre de coder. rendez-vous dans le menu « *Insertion* » puis cliquez sur « *Module* » :



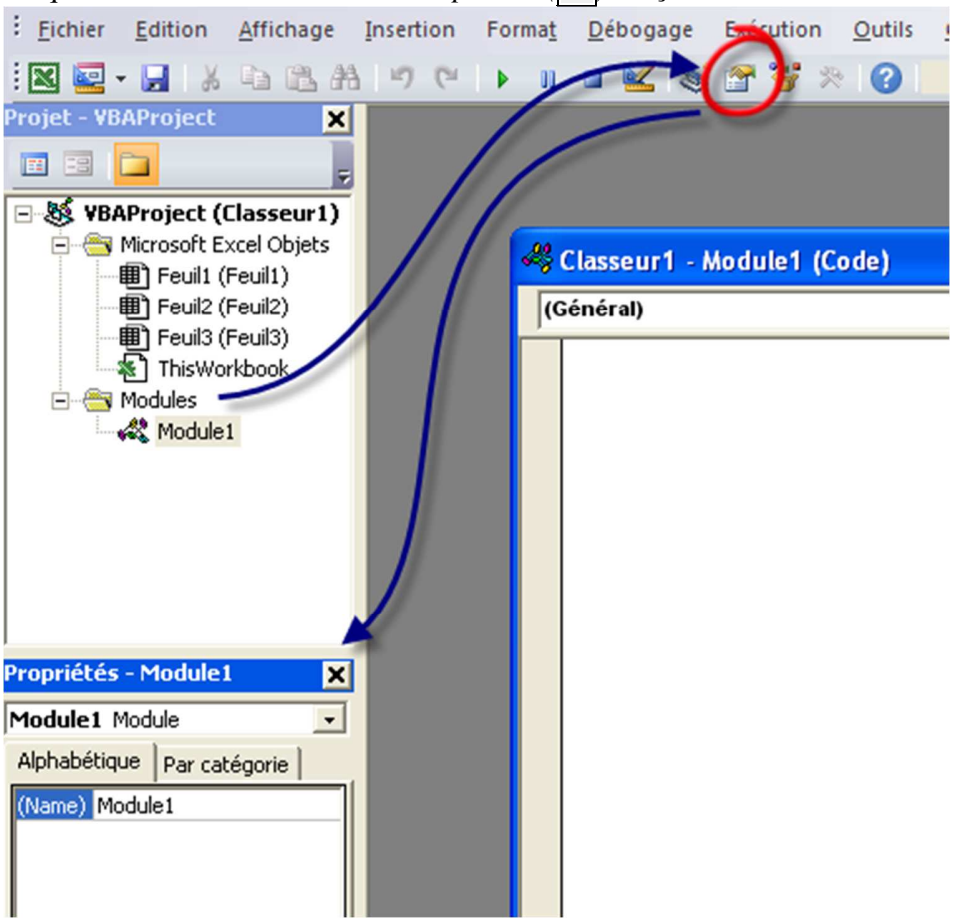
Remarquez l'apparition du dossier Module dans votre projet. C'est ici que seront rangés tous vos modules. Nous allons pouvoir commencer à coder.

STOOOOP ! Et si j'ai 40 modules ? Je vais avoir 40 fichiers dans mon dossier mais s'ils s'appellent Module1, Module2, Module3... je ne vais jamais m'en sortir !
Exact. Cher Zéro, vous soulevez la question du renommage du module, une démarche assez tordue.

Soit, je vais vous expliquer comment renommer un module.

Tout d'abord, sélectionnez votre module.

Cliquez sur le bouton « *Fenêtre Propriétés (*`F4`*)* » : ça affichera le menu des propriétés du module :



Renommez le module comme à vos envies, puis fermez cette petite sous-fenêtre au moyen de la petite croix.

Voilà, vous êtes fin prêts au codage. Petite précision tout de même : pour revenir au tableur depuis MVBA, il faut appuyer sur `Alt` + `F11`.

Codez votre première macro !

Dans toute cette seconde partie, nous allons coder en VBA, je pense que vous l'avez compris. Afin d'obtenir la coloration de mes codes d'exemples, j'ai choisi d'écrire ici mes codes entre des balises de coloration pour VB.NET. C'est un langage proche du VBA par sa syntaxe, et c'est pourquoi je m'en sers afin de colorer mes codes (ce qui est plus agréable à lire).
Pour créer une macro, vous pouvez soit faire travailler l'enregistreur de macros, et dans ce cas du code VBA sera généré selon vos désirs, soit la coder à la main.

Comme nous sommes sur le Site du Zéro et qu'ici, on est plutôt orienté « code », nous allons... coder.

Déclarer sa macro

Une macro porte un nom que vous lui donnez. Les espaces et les accents sont interdits.

- MAUVAIS = ma première macro.
- BIEN = ma_premiere_macro.

Chaque macro doit être codée entre les mots-clés **Sub** et **End Sub**.

Syntaxe

Voici sans plus tarder la syntaxe d'une déclaration :

```
Sub nom_de_la_macro ()



End Sub
```

Dans votre éditeur de code, écrivez seulement la première ligne `Sub nom_de_la_macro ()` (en remplaçant « *nom_de_la_macro* » par ce que vous voulez, tant que ça respecte les règles énoncées ci-dessus) et appuyez sur `Entrée` : `End Sub` a été généré tout seul !
Par exemple (nom repris du chapitre précédent) :

```
Sub gras_rouge ()



End Sub
```

Que s'est-il passé ?
Retournez à votre tableur, et dessinez un objet auquel vous appliquez votre macro (voir chapitre sur les macros).

Que constatons-nous ? Votre macro est reconnue, et ce, sans toucher l'enregistreur ! 😁

Les commentaires

Dans le code, vous avez la possibilité de placer des commentaires.

Ils n'auront aucune influence lors de l'exécution du code et servent simplement à vous repérer.

Un commentaire commence par une apostrophe '.

Si vous ne vous êtes pas trompés, le commentaire devrait apparaître en vert.

```
Sub ma_macro()



'Ceci est un commentaire.

'Il sert à vous retrouver dans le code.

'On placera le code de la macro ici. :)



End Sub
```

Pfiou ! 😅 Je crois que vous êtes prêts pour passer au concret.

VBA ..... > OK

Description de l'interface de développement ..... > OK

Organisation du projet ..... > OK

Première macro ..... > OK

Commentaires ..... > OK

C'est bon, vous pouvez passer à la suite ! 

Le VBA : un langage orienté objet

Après un premier chapitre d'introduction au VBA, il est temps de rentrer dans le concret. 

Si nous résumons, vous savez déclarer une macro et placer un commentaire dans un code.

Nous avons même constaté qu'une liaison a été établie entre votre macro et le tableur à "proprement parlé" puisque, sans passer par l'enregistreur de macro, vous pouvez affecter votre bout de code à un objet sur votre quadrillage.

Ce chapitre introduit des notions fondamentales pour cette troisième partie du cours. Allez-y à votre rythme, mais ne brulez pas les étapes. Si vous ne comprenez pas tout, même après relectures, passez à la suite. Vous aurez peut-être un « déclic » en pratiquant. Il est vrai que cette partie est fondamentale par la suite donc il est préférable de la maîtriser, ça va de soi. Le fait de passer à autre chose (chapitre suivant) vous permettra peut-être de comprendre puisque les connaissances acquises seront utilisées d'une manière différente.

Orienté quoi ?

Le VBA est un langage orienté objet. On dira également que vous faites de la Programmation Orientée Objet (POO). Ces mots n'ont probablement aucun sens pour vous, à moins que vous n'ayez déjà fait du C++ ou encore du Java.

Nous allons tenter d'étudier le concept en lui-même, puis en suite de l'appliquer à notre problème.

Dans la vie courante, vous reconnaissez un objet parce qu'il a un état physique, il est visible et vous pouvez le toucher. Une brosse à dents est un objet, un verre est un objet, un ordinateur en est un également... bref, la liste est longue.

L'objet peut être reconnaissable grâce à sa couleur, par exemple, mais vous pouvez aussi effectuer des actions dessus.

Nous allons prendre comme exemple votre maison (ou appartement, ne me dites pas que vous habitez dans un bateau... même si ça ne change pas grand-chose). Une maison est caractérisée par **ses propriétés** : elle a

une année de construction, une couleur... mais on peut aussi y faire beaucoup d'action : *Nettoyer*, *Regarder la télé* ... on parle alors de **méthodes** .

À partir de ces propriétés et méthodes, vous pouvez imaginer plein de maisons différentes, en faisant varier le nombre de pièces, par exemple. Les propriétés permettent d'identifier la maison, de la caractériser, de la singulariser. Les méthodes forment toutes les actions que l'on peut exécuter à partir de cet objet.

Toutes ces maisons ont donc été fabriquées à partir d'un plan. On parle d'une **classe**.

Lorsque vous fabriquez un objet à partir d'une classe, on dit que vous faites **une instance de classe**.

M@teo21 a une belle image pour ceci : imaginez un architecte qui dessine un plan de maison. Le plan correspond ici à ma classe et les maisons aux objets : en effet, à partir du plan, vous pouvez bâtir autant de maisons que vous le voulez !! J'ajoute que la définition de toutes les maisons de l'Univers, même imbriquées dans des classes différentes s'appelle **une collection d'objets**. 

Une classe porte le nom mis au pluriel des objets qu'elle regroupe. Ainsi, toutes vos maisons peuvent être regroupées autour de la classe *Maisons*
La maison : propriétés, méthodes et lieux

Continuons avec notre exemple de la maison.

Vous êtes dans votre maison et vous voulez prendre un bain (c'est une méthode), vous allez donc devoir vous rendre dans la salle de bain. Pour cela, il y a un ordre à respecter. Vous devez d'abord trouver la ville dans laquelle se trouve la maison, puis l'adresse précise et enfin trouver la salle de bain.

Puisque toutes les villes se ressemblent, nous pouvons considérer la classe Villes. De là, vous trouvez votre ville à vous, qui est une instance de Villes, ou un objet issu de Villes. Il en est de même pour la classe Maisons. Des maisons, il y en a des tonnes, mais la vôtre se distingue parce que c'est votre maison.

L'itinéraire à suivre est donc le suivant :

Ville > Maison > Salle de Bain > Bain

En code VBA, cet itinéraire se précise en partant du plus grand conteneur ; ici, la ville contient la maison, qui contient la salle de bain, et il y a la baignoire que nous désirons.

C'est comme les poupées russes : la ville est la plus grosse poupée qui contient toutes les maisons.

Les lieux et objets sont séparés par un point. Le code serait donc ceci :

```
Villes("Reims").Maisons("Ma_Maison").Salle_de_bains("Bain")




' Dans la classe Villes, votre ville se distingue des autres par son nom : Reims.

' Reims est un objet créé à partir de la classe Villes, qui contient aussi bien Paris que Bordeaux.
```

Vous accédez ainsi à l'objet "Bain". Entre parenthèses et guillemets, vous donnez des précisions. En effet, la baignoire se différencie des autres parce qu'elle permet de prendre un bain, vous ne pourriez pas construire un objet "Lavabo" à partir de la classe "Salle_de_bain" pour faire un bain.

Nous pouvons même rajouter une méthode à la fin, puisque vous désirez vous laver :

```
Villes("Reims").Maisons("Ma_Maison").Salle_de_bains("Bain").Frotter_le_dos
```

Et si vous désiriez vous laver les mains, on aurait pu créer ce fameux objet Lavabo, toujours issu de la classe Salle_de_bains

Tout ceci n'est que schéma bien sûr, mais la syntaxe correspond à celle d'un vrai code VBA. Vous prenez donc l'objet crée à partir de la classe *Salle_de_bain*, vous prenez une instance de la classe *Baignoire*.

Retenir :

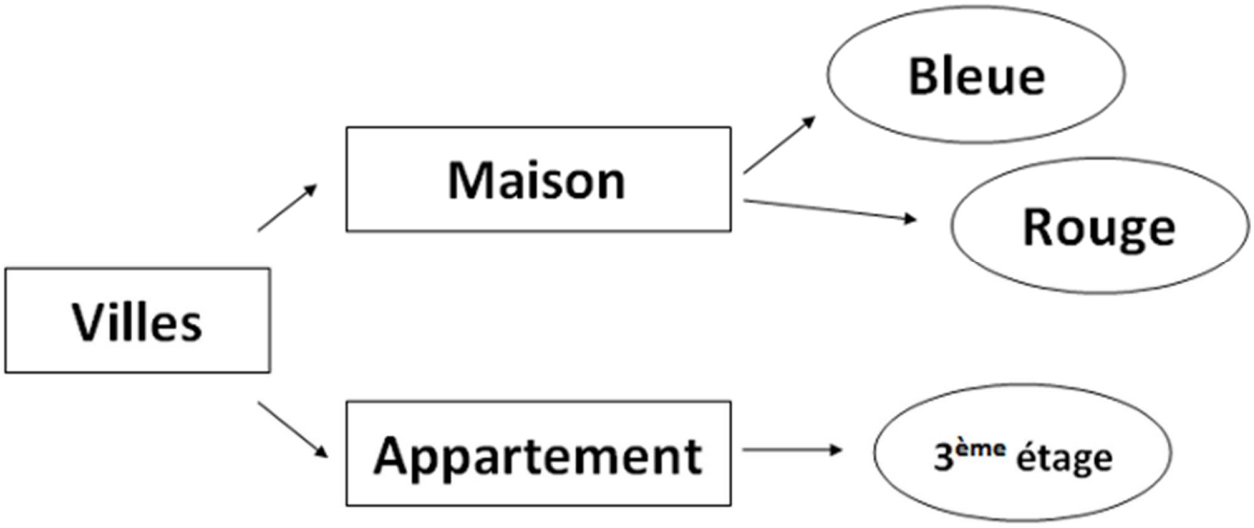
L'accès aux objets se fait comme suit :

nom_de_la_classe("Nom de l'instance de cette classe")

La POO en pratique avec la méthode Activate

Maintenant, il est temps de tester la POO en pratique, donc dans Excel (parce que les maisons, c'est bien, mais nous nous éloignons).

Je propose toutefois un schéma qui va aider à comprendre la suite :

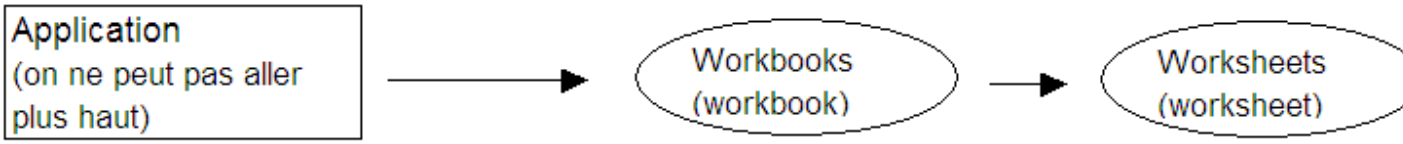


On voit, par exemple, que les couleurs sont des instances de la classe *Maison* : ils ont quelque chose en commun, comme la forme de la maison.

De même pour l'appartement du 3ème étage, qui est une instance de la classe *Appartement*. Le tout est contenu dans un grand objet : la ville.

Pour Excel, c'est un peu la même chose : le big des big objets, c'est Application, qui désigne l'application Microsoft Excel.

Lui-même contient la classe Workbooks, qui regroupe tous les classeurs Workbook ouverts. Et Workbook contient la classe Worksheets, qui contient toutes les feuilles Worksheet du classeur désigné. Un schéma pour mieux comprendre :



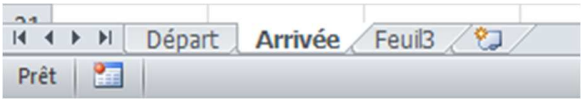
La POO en pratique

Nous allons faire nos débuts en POO avec la méthode **Activate**, qui active (qui vous amène) là où vous lui demandez.

Par exemple, je veux aller de la première feuille à la deuxième. Il va falloir donc nommer notre classeur et deux feuilles, afin de donner un itinéraire.

Enregistrez votre classeur en le nommant "*Essai*". Renommez une première feuille "*Départ*" et l'autre "*Arrivée*". (l'explication se trouve dans la seconde annexe).

Vous obtenez quelque chose dans ce genre :



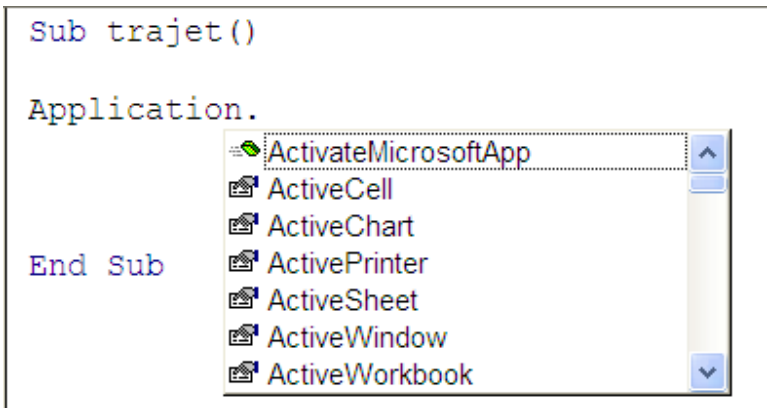
Placez-vous sur la feuille Départ, ouvrez la fenêtre de VBA, créez un nouveau module.

Maintenant, réfléchissons à l'itinéraire. On part de l'application, pour aller vers le classeur "Essai" et vers la feuille "Arrivée".

Le code serait donc :

```
Sub trajet()



Application.Workbooks("Essai").Worksheets("Arrivée").Activate



'On part de l'application vers l'instance Essai de la classe Workbooks

'ensuite, on va à l'objet Arrivée de la classe Worksheets



End Sub
```

Notez que le logiciel peut vous proposer une liste de classes :



Toutefois, on peut le raccourcir : c'est comme avec les pièces de la maison, si vous êtes dans la maison, il est inutile de préciser qu'il faut aller dans cette ville et à l'adresse de la maison, puisque vous y êtes déjà.

Ici, vous êtes bien sûr l'application Microsoft Excel (logique) et vous êtes aussi sur le classeur "Essai".

Le bon code est donc :

```
Sub trajet()



Worksheets("Arrivée").Activate



End Sub
```

Il ne vous reste plus qu'à aller sur la feuille "Départ", d'y dessiner un rectangle, d'affecter votre macro et de cliquer dessus, vous serez "téléporté" vers la feuille "Arrivée" 

A retenir

La classe **Workbooks** désigne tous les classeurs ouverts.

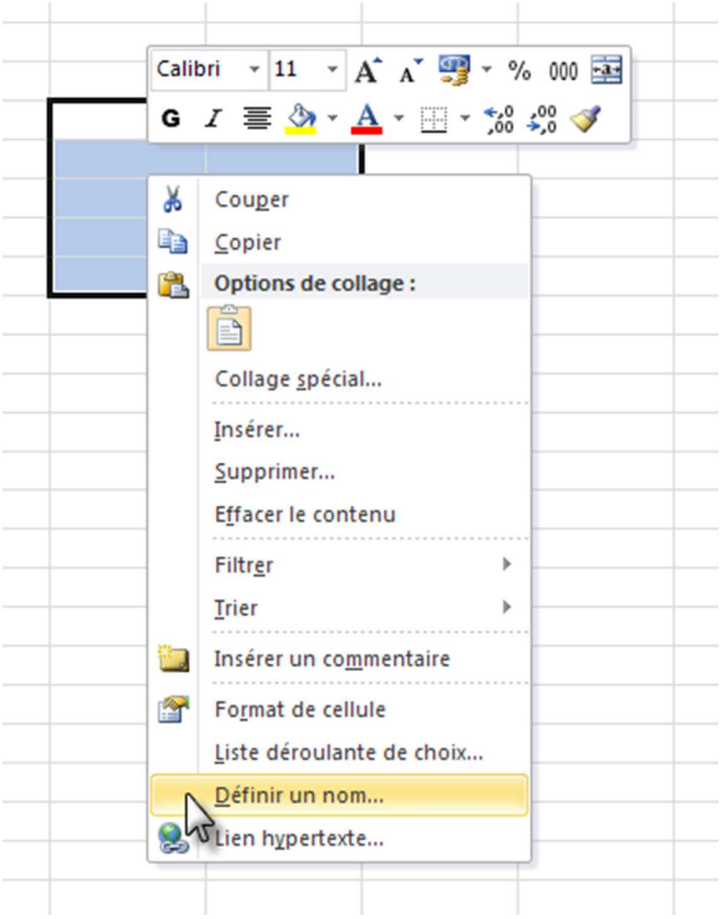
La classe **Worksheets** désigne toutes les feuilles du classeur actif.

D'autres exemples

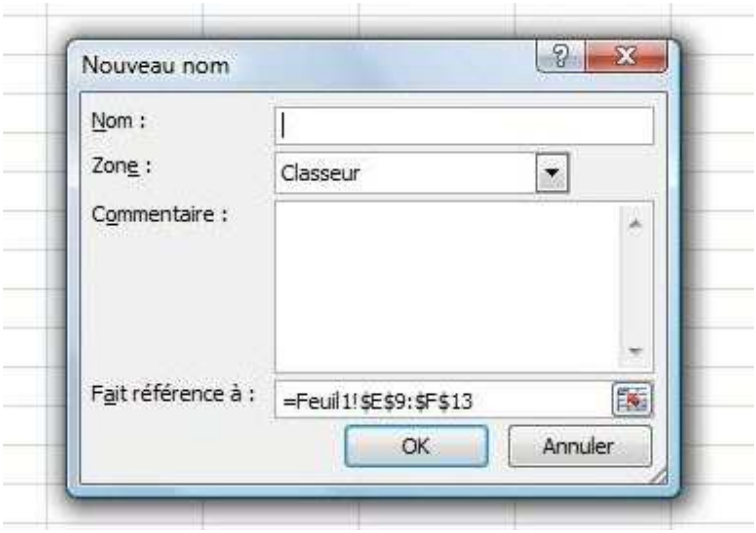
Nous allons sélectionner une plage de cellules en tapant un code VBA (bien que nous pourrions le faire par le biais de l'enregistreur de macros, mais cette partie deviendrait donc dépourvue d'utilité )

Tout d'abord, il serait bien de ne pas avoir à taper des plages dans un code via les coordonnées : il y a tellement de chiffres, de lettres, de caractères (guillemets, deux points) qu'on s'y perdrait. Je vais donc vous apprendre à ... nommer une plage de cellules !  D'ailleurs, vous pourrez trouver son utilité dans les formules, bref, ça clarifie un code. Ainsi, vous appellerez la plage par son nom.

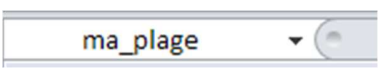
Voici comment on nomme une plage de cellule. Sélectionnez une plage de cellule, quelle qu'elle soit. Lorsque cette plage est sélectionnée, faites un clic droit et cliquez sur « *Définir un nom* » :



Une fenêtre s'ouvre, il suffit de remplir le champ « *Nom* » et de cliquer sur « *OK* » :

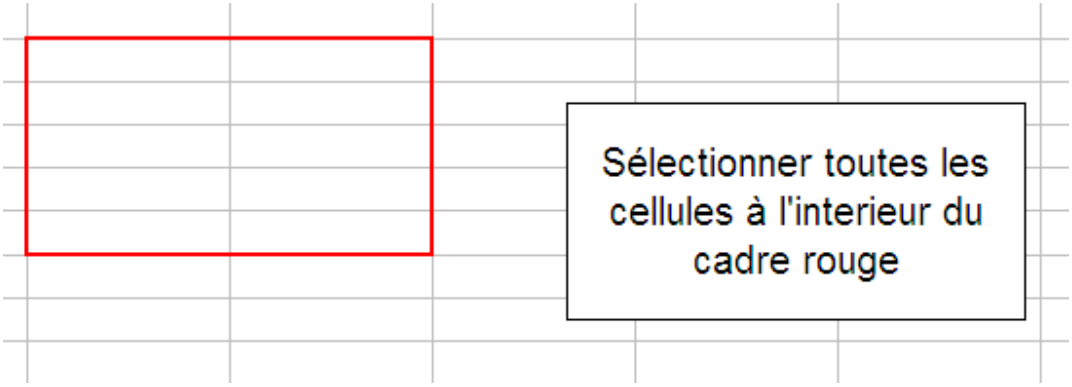


Vous remarquez que, à gauche de la barre de formule, apparait le nom de votre plage. C'est ce nom de plage que vous pouvez utiliser.



Vous pouvez retrouver votre plage à partir de cette liste déroulante en cliquant sur le nom de votre plage.

Mais revenons à notre feuille de calculs et préparons le terrain : mettons une bordure rouge autour de notre plage nommée et dessinons un rectangle, afin d'y appliquer la macro de sélection (que nous allons coder).



Aller, hop hop hop, ouvrez VBE ! On commence à coder :

```
Sub MaSelection()



' on placera le code ici



End Sub
```

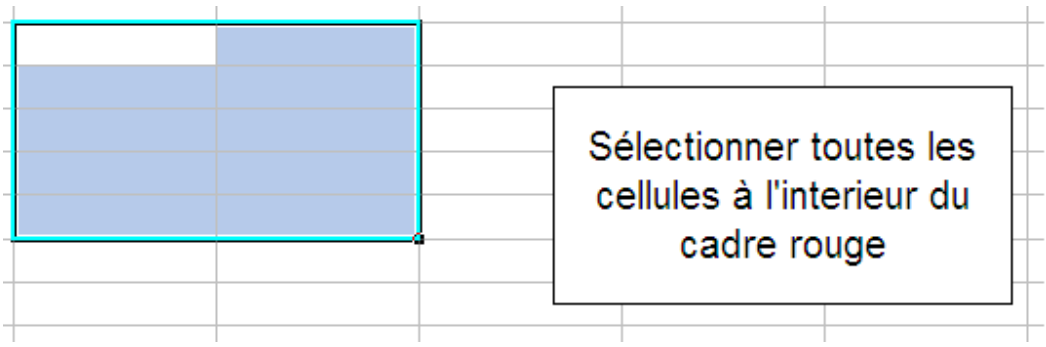
Rien de palpitant. Les cellules sont sous la tutelle de la classe **Range**, que ce soit une cellule ou une plage de cellules (jointes ou non). Nous allons utiliser également la méthode **Select**, qui sélectionne ce que vous lui demandez de sélectionner.

Je rappelle qu'il ne sera pas nécessaire de faire mention du classeur ou de la feuille de calculs active, puisque nous y sommes déjà (donc inutile de dire d'y aller).

Voici le code :

```
Sub MaSelection()



Range("Ma_Plage").Select



End Sub
```

Retour à notre feuille de calculs, où nous affectons la macro *MaSelection* au zoli rectangle . Après un clic dessus, la magie opère :



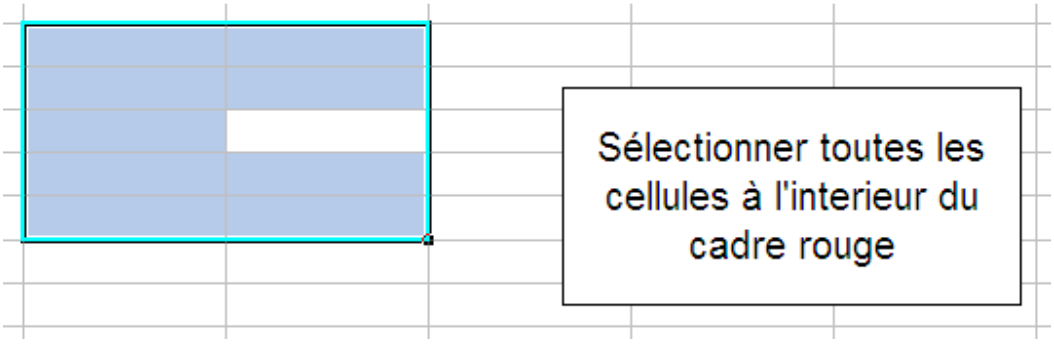
Vous rappelez-vous le nom et de la fonction particulière de la cellule de la sélection en haut à gauche, qui n'est pas sur fond bleu ? Mais oui, c'est la cellule active ! Pour rappel, si vous saisissez des données directement sur votre sélection, elles seront rentrées dans la cellule active.

Puis pour le *fun*, nous allons changer l'emplacement de cette cellule en VBA !

Il va falloir relever ses coordonnées ; dans mon cas, je prends C11, qui est une cellule de ma plage. Il va falloir utiliser la méthode **Activate** (vue ci-dessus) :

```
Sub MaSelection()



Range("Ma_Plage").Select

' on sélectionne la plage Ma_Plage, qui relève de la classe Range, à l'aide de la méthode Select



Range("C11").Activate

'la cellule C11 devient la cellule active de la sélection, avec la méthode Activate



End Sub
```

Et après un clic sur le rectangle, l'emplacement de la cellule active est effectivement modifié :



Voilà pour les méthodes. Il y en a beaucoup d'autres. Nous les verrons en temps voulu car chacune a une fonction bien particulière. L'idée était surtout de vous initier aux manipulations des méthodes.

Les propriétés

Bien, nous avons fait joujou avec la méthode **Activate**, histoire de mettre en application ces histoires de classes et d'instances. La technique vue ci-dessus est valable pour toutes les méthodes, ce qui signifie que c'est comme ça qu'on en applique une.

Maintenant, nous allons nous intéresser aux propriétés de l'objet (= instance).

Les propriétés : la théorie

Nous allons reprendre (et ce, pour encore un bout de temps), l'exemple de la maison. Si vous avez pigé tout ce qui a été dit avant (dans le cas contraire, n'hésitez pas à commenter ce chapitre en précisant les points obscurs), vous devriez être capables de dresser une petite liste des propriétés possibles d'une maison :

- Elle a une couleur : couleur
- Un nombre de pièces : nombre_de_piece

Voici comment nous pourrions accéder à ces propriétés (toujours en supposant que notre maison est bien issue de la classe Maisons et que l'objet se nomme Ma_Maison) :

```
Maisons("Ma_Maison").couleur = "verte"


'On dit que nous souhaitons accéder à l'objet Ma_Maison, de la classe Maisons


'nous accédons à la propriété couleur et nous lui affectons une valeur
```

Nous pouvons faire de même avec la propriété nombre_de_piece, car la syntaxe est la même :

```
Maisons("Ma_Maison").nombre_de_piece = 7
```

Vous remarquez que nous pouvons affecter différents types de valeurs :

Type de valeur	Exemple

Numérique	1, 5, 12 365 ...
	`Maisons("Ma_Maison").nombre_de_piece = 7`
	`'7 pièces par exemple`

Chaîne de caractères	Bonjour, a, jojo, 1234 (ici, 1234 sera lu comme une suite de caractères : 1, 2, 3, 4, et non plus comme un nombre) `Maisons("Ma_Maison").couleur= "Rouge"`
Booléen	TRUE, FALSE : le booléen ne peut prendre que deux valeurs, TRUE ou FALSE (vrai ou faux). Ainsi, nous pourrions imaginer le choix d'une maison avec étage ou de plain-pied.

Type de valeur	Exemple

	Si elle est avec étage, la propriété Etage pourrait valoir FALSE et si elle est de plain-pied, elle pourrait valoir TRUE.
	
	`Maisons("Ma_Maison").Etage = TRUE`
	
	`' Notre maison a au moins un étage`

Constante	La constante est... assez particulière, mais pourtant très utilisée. Une constante est une valeur qui ne change pas. Par exemple, plusieurs constantes peuvent être valables pour notre objet. Elles peuvent ici désigner le type de notre maison.
	
	`Maisons("Ma_Maison").Type = VILLA`
	
	`' Notre maison est une villa`
	
	`Maisons("Ma_Maison").Type = APPARTEMENT`
	
	`' Notre maison est un appartement`
	Les constantes propres à Excel commencent par le préfixe **xl**.

Les propriétés : la pratique

Il faut maintenant appliquer toute cette théorie à notre bon vieux VBA.

Afin de voir en une fois nos quatre types de propriétés, nous allons créer une macro qui répond aux consignes suivantes :

- - Taille du texte : 14
- - Police du texte : Euclid
- - Texte souligné
- - Texte en gras

Vous l'avez sans doute compris, nous allons travailler sur une sélection de texte.

Mais je ne sais pas situer une sélection de texte, moi ! 
Plus pour longtemps. Pour travailler sur la sélection, nous allons utiliser l'objet **Selection**. De plus, c'est l'objet **Font** qui gère les polices. Décortiquons les étapes à suivre :

La taille du texte

La taille du texte est contenue dans une propriété appelée **Size**. Cette propriété n'accepte que des valeurs numériques (la taille du texte). Un exemple juste parce que c'est le premier :

```
Sub taille()



Selection.Font.Size = 12



'nous modifions la taille du texte sélectionné à 12



End Sub
```

La police du texte

La police se trouve dans la propriété **Name**. Cette propriété attend une chaîne de caractères.

Le soulignement

Le soulignement est géré par la propriété **underline**. Cette propriété n'accepte que des constantes. Oui, ces mystérieuses constantes qui commencent par *xl*. Sachez que la valeur **xlUnderlineStyleSingle** applique un soulignement simple au texte. En effet, avec **xlUnderlineStyleDouble**, on peut appliquer au texte deux traits de soulignement.

Le gras

Le gras est une propriété à lui tout seul : **Bold**, qui signifie "gras" en anglais. Vous remarquerez d'ailleurs au fil du cours que le VBA a une syntaxe très proche de l'anglais. Bold n'accepte que deux valeurs : True ou False ; c'est donc un booléen. Si la propriété vaut True, le texte sélectionné sera mis en gras.

Dans le même genre, sachez qu'il existe la propriété **Italic**, qui est également un booléen et qui gère la mise en italique du texte.

Avez-vous su écrire la macro avec toutes ces indications ? Voici le corrigé :

```
Sub texte()



Selection.Font.Size = 14

Selection.Font.Name = "Euclid"

Selection.Font.Underline = xlUnderlineStyleSingle

Selection.Font.Bold = true
```

```
End Sub
```

Et si en plus de ça je veux mettre mon texte en italique, en couleur, et le centrer dans la cellule ? Je ne vais tout de même pas réécrire Selection.Font 50 000 fois !?

Et pourquoi pas ? Plus sérieusement, les concepteurs du VBA ont pensé à ce cas là bien avant vous. Ils ont même une solution de feignants (les programmeurs sont des feignants, on ne le dira jamais assez )

L'idée est d'expliquer à VBA qu'on va travailler un certain temps sur Selection.Font. Pour ce faire, nous allons utiliser une nouvelle notion : la structure en `With ... End With`

Une alternative de feignants : With ... End With

Cette structure est utilisée pour faciliter les modifications des propriétés sur un même objet. Sa syntaxe est la suivante :

```
With nom_de_votre_objet



'on fait ici nos modifications sur les propriétés par exemple :

    .propriété_1 = valeur_1

    .propriété_2 = valeur_2

    .propriété_3 = valeur_3



End With
```

Ainsi, le code de la modification du texte équivaut à :

```
Sub texte()



With Selection.Font



    .Size = 14

    .Name = "Euclid"

    .Underline = xlUnderlineStyleSingle

    .Bold = true
```

```
End With



End Sub
```

On peut voir l'utilité de cette fonction lorsque l'on utilise des objets dont le nom est très long à écrire.

```
Sub test()



With Application.Workbooks("classeur1").Worksheets("feuil1").Range("A1")



    .propriété_1 = valeur_1

    .propriété_2 = valeur_2

    ...



End With



End Sub
```

Voilà, c'est fini pour la partie un peu théorique. Au menu du chapitre suivant : pause bien méritée !

En attendant, si vous en voulez encore, faites des tests. Il n'y a que comme ça que vous progresserez et la POO ne s'assimile pas du jour au lendemain. Il faut absolument avoir compris l'idée de ce concept, sans quoi il est inutile de poursuivre.

La sélection

Après une mise en bouche à la POO, nous allons faire une pause. Vous avez quand même ingurgité pas mal de notions dans les deux chapitres précédents. Nous n'allons pas pour autant glander. Ici, pause est synonyme de... pratique !

En effet, la théorie, c'est bien, mais il faut bien mettre les mains dans le cambouis à un moment où à un autre. Nous n'allons donc pas étudier de nouvelles notions propres à la POO mais tout simplement mélanger classes, objets et méthodes plus ou moins nouvelles afin de vous faire travailler la sélection.

Si vous avez suivi la première partie de ce cours, vous avez sans doute compris à quel point l'accent a été mis très tôt sur la sélection. C'est une notion importante et même en VBA. Gardons à l'esprit que VBA peut permettre d'échanger avec un utilisateur du tableur. Il faut donc bien pouvoir s'y repérer, non ? 

Sélectionner des cellules

Comme nous l'avons très rapidement vu au chapitre précédent, toutes les cellules du tableur sont des instances de la classe **Range**. Pour effectuer la sélection, il suffit d'appliquer à votre objet la méthode **Select**. Jusque-là, pas de surprise. La sélection de la cellule C5 peut donc se faire comme suit :

```
Sub tests_selection()



Range("C5").Select



End Sub
```

On peut aussi sélectionner plusieurs cellules les séparant par des virgules :

```
'Sélection des cellules C5, D4 et F8 :



Range("C5, D4, F8").Select
```

Votre objet peut aussi être une plage de cellules. Par exemple, le code ci-dessous sélectionne une plage de B2 à E8 :

```
Range("B2:E8").Select
```

Cells, une autre classe pour les cellules...

Les cellules sont des instances de Range, mais aussi de la classe **Cells**. Voyez plutôt ces deux codes comparatifs qui exécutent exactement la même chose :

Avec Range	Avec Cells
`Range("E3").Select`	`Cells(3, 5).Select`

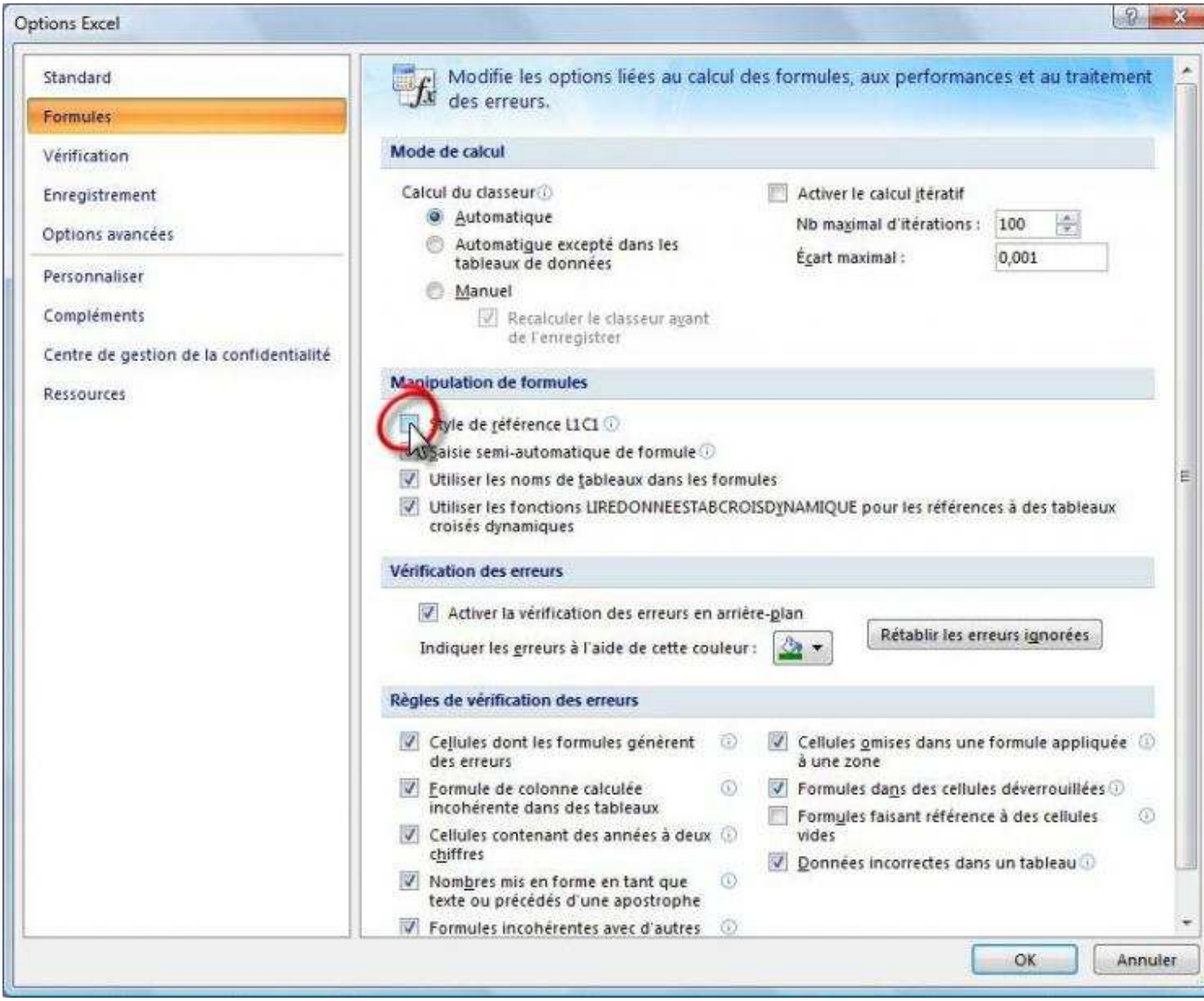
La syntaxe pour sélectionner une cellule avec Cells est la suivante :

```
Cells(ligne, colonne).Select
```

C'est exactement l'inverse de l'adressage que nous connaissons dans le tableur. D'abord la ligne, ensuite la colonne. Attention ! Ces deux valeurs doivent être numériques ! C'est-à-dire que vous devez récupérer le numéro de la colonne souhaitée.

Mais comment je fais pour connaître le numéro de la colonne ?

Cliquez sur le gros bouton Office en haut à gauche puis sur « *Option Excel* ». Dans la fenêtre qui s'ouvre cliquez à gauche sur « *Formules* » et dans la rubrique « *Manipulation de formules* », cochez la case « *Style de référence L1C1* » :



Vos références de colonnes sont maintenant des chiffres, et non des lettres

Décaler une sélection

Vous avez été traumatisés par les translations en maths ? Aïe pour vous. On va apprendre une technique qui sert à décaler une sélection à partir de la cellule active.

La cellule active est gérée par la classe **ActiveCell**. La classe nouvelle qui effectuera le décalage est **Offset**.

Offset attend deux arguments : le nombre de lignes puis le nombre de colonnes du décalage. Il ne faut pas se tromper dans le signe. Des exemples ? En voici :

```
Sub decalages()

```

```
'La cellule active ne bouge pas :



ActiveCell.Offset(0, 0).Select



'Décalage d'une ligne vers le bas et d'une colonne vers la droite :



ActiveCell.Offset(1, 1).Select



'Décalage d'une ligne vers le haut et de trois colonnes vers la gauche :



ActiveCell.Offset(-1, -3).Select



End Sub
```

Sélectionner des lignes

Les lignes du tableur sont des objets de la classe **Rows**. La sélection de la deuxième ligne du classeur actif peut se faire comme ceci :

```
Rows("2").Select
```

On utilise toujours la même méthode (Select). C'est juste la classe qui change 

Tout comme pour les cellules, on peut aussi sélectionner une plage de lignes contiguës :

```
' Sélection des lignes 2 à 5 incluses :



Rows("2:5").Select
```

Les lignes peuvent aussi être discontinues. Dans ce cas, la syntaxe est la suivante :

```
Range("2:2, 5:5, 6:8").Select
```

Les lignes 2, 5 et la plage des lignes 6 à 8 seront sélectionnées.

Hein ?? Mais tu avais dit que les lignes étaient gérées par Rows ! Pourquoi tu nous ressors Range ?
Lorsqu'on cherche à faire des sélections d'objets discontinus en VBA, on retourne vers la classe Range. C'est comme ça. On indique les plages que l'on souhaite sélectionner séparées par des virgules.

Sélectionner des colonnes

La méthode pour sélectionner des colonnes est la même que pour sélectionner des lignes. Nous allons juste utiliser la classe **Columns** qui s'occupe des colonnes du tableur Je pense que vous savez désormais sélectionner une colonne...

```
'Sélection de la colonne C :



Columns("C").Select
```

Vous pouvez aussi sélectionner une plage de colonnes :

```
'Sélection des colonnes C, D, E et F :



Columns("C:F").Select
```

Et comme pour les lignes, on se tourne vers la classe **Range** pour la sélection d'objets discontinus :

```
'Sélection des colonnes A, C et G :



Range("A:A, C:C, G:G").Select
```

Exercice : faciliter la lecture dans une longue liste de données

.

Le repos est terminé, on retourne dans les zones sombres du VBA ! Au programme : deux chapitres sur une notion fondamentale de tout langage de programmation qui se respecte : les variables !

Les variables 1/2

Après une longue introduction aux notions fondamentales de la POO, il est temps d'explorer les contrées sombres du VBA. Nous commencerons par les variables. Si vous connaissez d'autres langages de programmation, vous savez à quel point elles sont importantes. On les retrouve partout ! Mais bon, nous partons de zér0, donc vous ne savez pas ce que c'est.

N'oubliez pas que les variables sont fondamentales, aussi deux chapitres y seront consacrés. Ne brûlez pas les étapes et allez-y doucement.

Qu'est ce qu'une variable ?

Pour faire un programme, on a besoin de stocker des informations pendant un certain temps pour ensuite le réutiliser. Pour cela, on va utiliser un espace de stockage pour stocker l'information que l'on souhaite retenir.

Pour vous aider je vous donne un exemple : on demande à l'utilisateur d'entrer son prénom et on va le stocker dans une variable prenom. Par la suite, dans l'exécution du programme, on fera référence à la variable prenom pour afficher son

prénom. Sachant que le prénom est différent selon l'utilisateur et que nous pouvons lui demander de changer de prénom, on appelle cette notion *variable* puisqu'elle peut changer au cours du temps.

Pour stocker une variable, on va avoir besoin d'un espace de stockage que l'on demande à l'application. Elle ne refusera jamais sauf s'il n'y en a plus, mais c'est très rare. En revanche, on ne va pas demander n'importe quel espace. Pour stocker du texte, on demande une variable de type "texte" et pour un nombre une variable de type "nombre". De ce fait, si vous essayez de stocker un nombre dans une variable texte, l'exécution ne se fera pas correctement.

Nous allons étudier les différents types plus tard dans ce chapitre.

Comment créer une variable ?

Nous savons maintenant ce qu'est une variable. Il est temps d'apprendre à les créer ! On parle alors de déclaration de variable. Il existe deux méthodes de déclaration : explicite et implicite. Étudions les deux méthodes.

Déclaration explicite

La déclaration de variable explicite est très simple, voici la syntaxe :

```
Dim ma_variable_explicite [As type]
```

Étudions ce bout de code.

Pour déclarer une variable, on écrit en premier Dim. Vient ensuite le nom de notre variable qu'il faut un minimum explicite, mais pas non plus une phrase. Enfin, le type de la variable. Il est ici entre crochets puisqu'il n'est pas obligatoire. S'il n'est pas indiqué, la variable est de type Variant par défaut (nous allons voir les différents types ultérieurement).

Revenons sur le nom de la variable. Une variable que vous appelleriez x n'est pas explicite et difficile à repérer dans un programme. Vous savez peut-être que c'est un nombre, mais si vous revenez sur votre programme par la suite, la signification vous aura peut-être échappée. Dans l'extrême inverse, on a aussi une variable appelée nombre_le_plus_grand_de_toute_la_serie. C'est certain qu'il est explicite, par contre, il ne faudra pas se tromper lorsque vous avez à y faire référence. Plus il y a de caractère et plus la probabilité de se tromper est grande. S'il y a un seul caractère différent, le programme ne reconnaitra pas la variable.

Autrement dit, les noms du genre :

```
Dim temp

Dim data

Dim x
```

sont à proscrire absolument. 

Il y a des règles à respecter lorsque vous déclarez une variable :

- Elle doit commencer par une lettre
- Elle ne doit contenir que des lettres, des chiffres et le caractère de soulignement, aussi nommé *underscore* (touche 8 du clavier azerty français _). Les espaces sont interdits !
- Elle est limitée à 40 caractères.

- Elle ne doit pas être identique à un mot réservé.

Qu'est ce que la liste des mots réservés ?

Cette liste contient tous les mots réservés au bon fonctionnement de VBA, c'est-à-dire les mots-clés du langage. *Dim* en est par exemple un puisqu'il sert à déclarer une variable. On ne peut pas non plus attribuer à une variable le nom Sub. Ce mot est réservé pour déclarer les procédures. C'est la même chose avec les noms de fonctions et tous les mots réservés à VBA. Si vous ne savez pas s'il est réservé, un message d'erreur apparait lors du codage, il suffit alors de choisir un autre nom.

On peut ensuite attribuer une valeur à notre variable :

```
ma_variable_explicite = 10
```

Déclaration implicite

La déclaration de variable explicite était simple, l'implicite l'est encore plus ! Voici la syntaxe :

```
ma_variable_implicite = 6
```

Vous avez compris le truc ? Et bien, la variable est utilisée directement. Elle n'a pas été déclarée avec Dim. Le programme demande automatiquement de créer cette variable, qui sera par défaut de type Variant et aura pour valeur 6.

Cette approche peut poser problème et engendrer des erreurs difficiles à détecter si vous n'orthographiez pas correctement le nom de la variable dans une commande ultérieure du programme.

Par exemple, si dans la suite de votre programme vous faites référence à ma_variabl_implicite au lieu de ma_variable_implicite, vous savez ce que cela signifie, mais VBA l'ignore. Il va supposer que ma_variabl_implicite est une nouvelle variable et la créé en tant que telle. L'ancienne variable, ma_variable_implicite, est toujours là, mais elle n'est plus utilisée. Vous avez à présent deux variables différentes alors que vous pensez n'en avoir qu'une seule. Cela peut causer d'énormes problèmes qui peuvent prendre beaucoup de temps à être résolus.

Quelle méthode de déclaration choisir ?

Là, c'est à vous de voir. Il y a des avantages et inconvénients dans les deux types. La méthode à utiliser dépend de vos préférences personnelles. Le codage est souvent plus rapide en utilisant la déclaration implicite parce que vous n'avez pas à définir à l'avance vos variables avant de les utiliser. Vous utilisez des variables dans les commandes et VBA s'occupe du reste.

Cependant, comme nous l'avons vu, cela peut engendrer des erreurs à moins que vous ne possédiez une très bonne mémoire et sachiez exactement ce que vous faites. Une déclaration implicite peut aussi rendre votre code plus difficile à comprendre pour les autres.

Obliger à déclarer

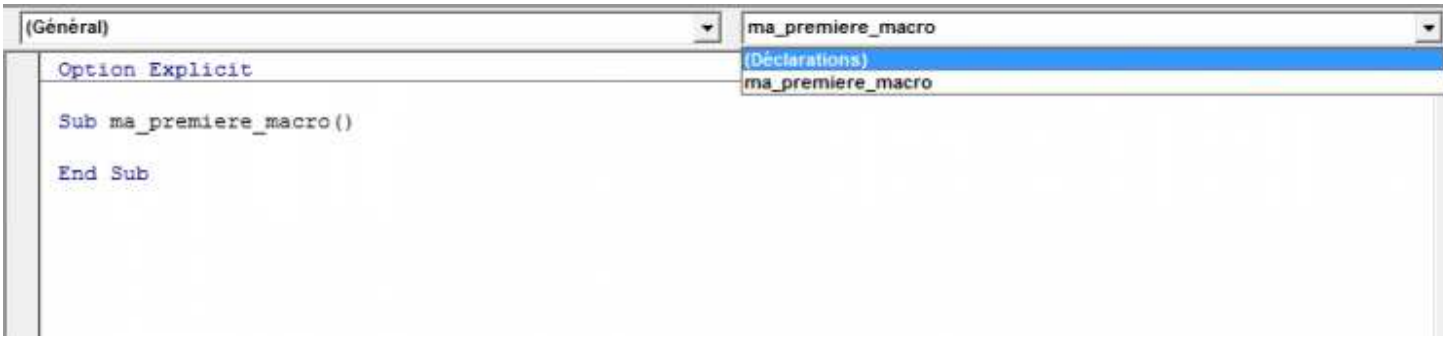
Si vous utilisez les deux méthodes, vous pouvez vous obliger à déclarer votre variable grâce à la fonction `Option Explicit`. Cela permet de ne jamais déclarer une variable de façon implicite. De cette manière, une variable non déclarée dans un code provoque une erreur dans la compilation du code et vous indique directement quelle variable est mal orthographiée ou non déclarée à cause d'un oubli.

Pour utiliser cette fonction, il faut l'introduire dans **TOUS** les modules dans lesquels vous souhaitez obliger les déclarations de variables, et ce, avant le code des procédures. Si vous regardez un module dans la fenêtre de l'éditeur Visual Basic, vous verrez une liste déroulante appelée *(Général)* dans le coin supérieur gauche de la fenêtre du module et une liste déroulante appelée *(Déclarations)* dans le coin supérieur droit de la fenêtre

du module. Cliquez sur *(Déclarations)* et vous irez directement dans la section des déclarations. Saisissez la commande suivante :

```
Option Explicit
```

Après avoir saisi cette commande, une ligne se place en dessous pour confirmer la présence de la fonction. Maintenant, chaque variable doit être déclarée. Essayez par vous-même d'exécuter un code sans déclarer une variable, une erreur arrêtera la compilation.



Que contient ma variable ?

Vous savez maintenant déclarer une variable et lui attribuer une valeur. Mais comment retrouver la valeur de cette variable et l'afficher à l'écran pour vérifier ? Pour cela, on va utiliser la fonction **MsgBox**. Cette fonction permet d'afficher ce que l'on veut sous forme d'une fenêtre qui s'ouvre dans le classeur concerné et qui bloque temporairement l'exécution de la procédure. Pour continuer la procédure, il faut cliquer sur "Ok".

Cette fonction sera utilisée abondamment dans la suite du cours. C'est pour ça que l'on va l'étudier ici.

Lorsque vous écrivez du code dans l'éditeur, vous pouvez exécuter ce code soit à l'aide de l'icône dans la barre d'outils ou en pressant la touche F5. Le code s'exécute alors dans le classeur actif. Dans notre cas, la boite de message (MsgBox) va s'ouvrir dans le classeur actif. Voici un exemple de boite de dialogue :

Pour afficher ce message, le code est très simple !

```
Sub bonjour()



 MsgBox "Bonjour"

```

```
End Sub
```

Cette syntaxe est nouvelle pour vous. Nous aurons l'occasion de reparler des fonctions plus loin dans le cours. Faîtes-moi confiance pour le moment et contenez-vous d'avaler cette histoire de MsgBox comme une recette de cuisine. 

A noter que tant que vous ne cliquez pas sur Ok pour terminer l'exécution du code, vous ne pouvez modifier votre code.

Je ne sais pas si vous avez fait votre feignant comme moi, mais si vous ne mettez pas les majuscules à MsgBox, VBA s'en charge ! C'est le cas pour Dim aussi, mais aussi pour toutes les fonctions qui sont présentes dans VBA. Tout le monde le dit, les programmeurs sont des feignants ! Ou pas...

Et pour afficher la valeur d'une variable alors ?

Pour afficher la valeur d'une variable, on va simplement attribuer à la fonction notre variable.

```
MsgBox ma_variable
```

Il n'y a pas besoin de mettre des guillemets, ceux-ci servent pour afficher du texte. Si vous en mettez autour de votre variable, la fonction affiche le nom de votre variable et non la valeur de celle-ci.

```
Sub test()



Dim ma_variable As String

ma_variable = "Bonjour"



MsgBox "Cette phrase affiche la valeur de ma variable : " & ma_variable



MsgBox "Cette phrase affiche le nom de ma variable : " & "ma_variable"



End Sub
```

Ce code affiche dans un premier temps :

Citation : MsgBox

Cette phrase affiche la valeur de ma variable : Bonjour

Puis dans une seconde boite de message :

Citation : MsgBox

Cette phrase affiche le nom de ma variable : ma_variable

Vous remarquez que pour mettre du texte et une variable dans la même phrase, vous pouvez concaténer les deux chaînes de caractères. Concaténer signifie mettre bout à bout. Pour cela on utilise le et commercial &. Il doit être présent à chaque fois que deux éléments différents sont adjacents. Un élément peut être une chaîne de caractère entre guillemets ou une variable.

C'est tout ce que je voulais vous faire découvrir sur cette fonction qui est très utilisée. On verra dans la suite du cours que l'on peut personnaliser ces boites de dialogue.

Types de variables

Un type de données peut être attribué à une variable ce qui permet de déterminer le type d'information que la variable peut stocker. Cette décision peut avoir un effet sur l'efficacité de votre code. Si aucun type de données n'est spécifié, le type par défaut est Variant.

On rappelle que pour attribuer un type à une variable, il faut faire suivre le type lors de la déclaration de variable.

```
Dim ma_variable As String
```

Variant

Une variable de type Variant peut stocker tous les types de données, par exemple, du texte, des nombres, des dates ou bien encore d'autres informations. Elle peut même stocker un tableau entier. Une variable Variant peut modifier librement son type pendant l'exécution du code alors que ceci est impossible à réaliser avec un autre type de données (par exemple String).

Vous pouvez utiliser la fonction **VarType** pour trouver le type de données que contient une variable Variant.

```
Sub TestVariables()



' On déclare implicitement une variable qui est du texte.

ma_variable = "le_site_du_zéro"



'On affiche le type de la variable

MsgBox VarType(ma_variable)



'On change la valeur de la variable pour mettre un nombre

ma_variable = 4



'On affiche le nouveau type de la même variable
```

```
MsgBox VarType(ma_variable)



End Sub
```

La boite de dialogue affichera d'abord la valeur 8 puis la valeur 2.

Mais, je croyais que la fonction nous renvoyait le type de la variable ?
C'est le cas, sauf qu'au lieu de vous le donnez en toutes lettres. On vous donne un chiffre qui correspond à un type. Voilà le tableau des correspondances :

Valeur de retour	Type
0	Empty
1	Null
2	Integer
3	Long
4	Single
5	Double
6	Currency
7	Date/Time
8	String

Par la même occasion, je viens de vous présenter la liste de tous les types de variables que l'on peut renvoyer pour une variable.

Même si Variant stocke tout type de données, il n'est pas indifférent à ce qu'il contient. En effet, vous avez vu dans l'exemple précédant que Varient identifie le type de données. Ainsi, si vous essayez d'effectuer des calculs avec une variable de type Variant mais contenant du texte, une erreur apparait. Il existe une fonction pour savoir si la variable est de type numérique ou non et donc ainsi faire des calculs. Cette fonction est **IsNumeric** et fonctionne comme la fonction VarType. Si la variable comprend un nombre, la fonction renvoie Vrai, sinon elle renvoie Faux.

```
Sub TestNumeric()



'On déclare une variable implicitement ayant du texte

ma_variable = "zozor"
```

```
'On affiche le résultat de la fonction

MsgBox IsNumeric(ma_variable)



End Sub
```

Cet exemple affiche la valeur Faux.

Reprenons la liste des types de variables que nous avons vues dans le tableau précédent. Il y a en fait des types qui peuvent se regrouper. On va détailler ces différents types et dire à quoi ils servent.

Les types numériques

On va ici détailler les types numériques : Byte, Integer, Long, Single, Double, Currency. Tous ces types de variables permettent de stocker des nombres.

Pourquoi avoir créé autant de types sachant qu'un nombre est un nombre non ?
Il est vrai qu'on pourrait avoir un seul type pour les nombres. Mais le chiffre 1 ne prend pas la même place en mémoire que le nombre 23954959,45593. C'est pour cela que différents types ont été créés. En effet, si la mémoire monopolisée par une variable est faible, alors le programme tournera plus vite. C'est logique après tout, il aura moins de données à analyser.

Voici un tableau qui récapitule les différents types numériques (la plage représente la plage de dispersion dans laquelle les valeurs de la variable doivent être comprises) :

Nom	Description	Plage	Caractère de déclaration
Byte	Contient un nombre entier (sans partie décimale = nombre après la virgule) sur 1 octet	0 à 255	Aucun
Integer	Contient un nombre entier sur 2 octets	-32 768 à 32 767	%
Long	Idem Integer sur 4 octets	- 2 147 483 648 à 2 147 483 647	&
Single	Contient un nombre en virgule flottant (partie décimale variable) sur 4 octets	-3,402823E38 à 1,401298E-45 (valeurs négatives) 1,401298E-45 à 3,402823E38 (valeurs positives)	!
Double	Idem Single sur 8 octets	-1,79769313486232E308 à -4,94065645841247E-324 (valeurs négatives) 4,94065645841247D-324 à 1,79769313486232D308 (valeurs positives)	#

Nom	Description	Plage	Caractère de déclaration
Currency	Nombre sur 8 octets avec une partie décimale fixe	-922337203685477,5808 à 922337203685477,5807	@

Oulala, c'est quoi tous ces chiffres ?
J'avoue que tous ces chiffres ne sont pas très digestes. Il faut simplement retenir que pour de petits entiers on utilise Integer ou Byte si l'on est sûr que c'est un petit nombre (comme pour l'age par exemple). Si on sait que l'entier sera très gros, on utilise Long. C'est la même chose pour Single et Double mais pour les nombres décimaux. Currency est moins souvent utilisé.

Euh, j'ai dû partir aux toilettes quand tu as expliqué le caractère de déclaration...
Non, c'est normal, je n'en ai pas parlé. J'allais y venir. Le caractère de déclaration est utilisé pour faciliter les déclarations de variable. Au lieu de mettre As Integer, on va simplement coller le caractère % à la variable pour la déclarer comme un Integer.

```
'Ces deux lignes exécutent la même chose

Dim ma_variable%

Dim ma_variable As Integer



'D'autres exemples de déclaration de variables

Dim age As Byte

Dim annee As Integer

Dim salaire_Zidane As Long

Dim un_pourcentage As Currency

Dim une_fraction As Single

Dim une_grosse_fraction As Double
```

le caractère de déclaration permet donc de gagner du temps dans le codage, mais n'est pas très explicite. Je ne peux vous conseiller une méthode, c'est selon votre façon de programmer.

On peut déclarer plusieurs variables sur une même ligne (avec un seul Dim. Mais pour le type, il faut le spécifier à chaque fois. S'il n'est pas spécifié après chaque variable, le type Variant est appliqué aux variables sans type spécifique.

```
Dim nom As String, prenom As String, age As Integer, enfants As Integer
```

Dans le tableau répertoriant les valeurs renvoyées par VarType, il n'y a pas le type Byte. Si une variable est de type Byte, la fonction renvoie 17.

Le type String

Si vous savez que votre variable ne sera que du texte, alors vous utilisez le type String. String signifie en anglais "Chaîne". C'est donc une chaîne de caractère que l'on va donner comme valeur.

```
Dim mon_texte As String
```

Ainsi, vous pouvez utiliser les fonctions sur les chaînes de caractères à ces variables.

Par défaut, la longueur de la chaîne de caractère est variable. On peut alors, pour économiser de la mémoire, fixer la longueur de la chaîne à un maximum. C'est au moment de la déclaration de variable que l'on effectue cette opération de cette façon :

```
Dim du_texte As String * 50
```

On utilise l'étoile (le signe multiplier informatique) suivie du nombre de caractères maximum. Dans notre exemple, la chaîne ne pourra dépasser 50 caractères.

Si la chaîne est inférieure au maximum, elle est comblée avec des espaces. A l'inverse, si elle est trop longue, elle est coupée après le 50ème caractère et le reste est perdu.
Il existe aussi un caractère de déclaration pour ce type de variable qui est le dollar $.

A titre indicatif, une chaîne de longueur fixe peut comporter jusqu'à 65 500 caractères alors qu'une chaîne de longueur variable peut comporter jusqu'à 2 000 000 de caractères.

Les types Empty et Null

Vous avez vu dans le tableau des différents types qu'il reste encore des types inconnus. Nous allons en analyser deux de plus ici, il s'agit de Empty et Null. Le premier cité est attribué à une variable ne contenant rien. C'est le cas d'une variable qui est déclarée et à laquelle n'est affectée aucune valeur. On peut vérifier si une variable contient quelque chose ou si elle a été définie avec un type grâce à la fonction IsEmpty.

```
' On déclare une variable sans type donc un Variant

Dim ma_variable_vide



' On déclare une variable de type String (du texte)

Dim mon_texte As String



' On n'affecte aucune valeur ni à l'une ni à l'autre variable

' On affiche le type des deux variables



MsgBox VarType(ma_variable_vide)
```

```
MsgBox VarType(mon_texte)
```
Dans un premier temps, on observe que la variable est vide donc s'affiche 0. Dans le second temps, la variable a beau être vide, mais elle est déclarée comme une chaîne de caractère, s'affiche alors 8 pour String.

Il ne faut pas confondre une valeur vide à une valeur Null. C'est le deuxième type de variable expliqué dans ce point. Une variable Varient peut contenir une valeur spéciale appelée Null. La valeur Null est utilisée pour indiquer des données inconnues ou manquantes. Les variables ne sont pas initialisées à Null à moins que vous n'écriviez du code pour ce faire. Si vous n'utilisez pas Null dans votre application, vous n'avez pas à vous soucier de la valeur Null.

La valeur Null n'est pas égale à 0.
Il existe une fonction pour savoir si une variable est de type Null ou non, il s'agit de la fonction IsNull.

```
Sub TestNull()



' On crée une variable à laquelle on attribue la valeur Null (sans guillemet)

ma_variable = Null



' On teste le type de la variable

MsgBox IsNull(ma_variable)



End Sub
```
Voilà deux types de variables en plus dans votre besace.

Le type Date et Heure

On ne peut définir une variable avec le type Date dès la déclaration. Elle se "transforme" en type Date une fois que la valeur est égale à une date. Les dates et heures sont stockées sous forment de nombres en virgule flottante. La partie entière représente le nombre de jours écoulés depuis le 31 décembre 1899. La partie décimale représente les heures, minute et secondes exprimées comme une portion de 24 heures (0,5 = 12 heures).

Il existe une fonction permettant de savoir si la valeur d'une variable est une date. C'est la fonction IsDate.

```
la_date = "23-07-2010"

MsgBox IsDate(la_date)
```
Ce code affiche Vrai.

La date peut être renseignée avec différents séparateurs. Ici j'ai utilisé les tirets, mais on peut utiliser les points ou les slashs. L'année peut comporter que deux chiffres. Il faut faire attention à la configuration du format de date qui peut faire varier la reconnaissance de date. JJ-MM-AAAA ou MM-JJ-AAAA.

Le type Booléens

Vous savez ce qu'est un booléen ? Ce mot un peu barbare désigne en fait un type de variable qui prend 2 valeurs différentes : Vrai et Faux. L'avantage est que ce type de variable ne demande que 2 octets pour être stocké et très souvent utilisé en programmation. VBA utilise l'anglais, la variable de type booléen prends alors les valeurs True or False. Par contre, à l'affiche dans une boite de dialogue, Vrai et Faux sont affichés.

```
' Une variable qui vaut Vrai

ma_variable = True



' Une variable qui vaut Faux

ma_variable = False
```

La fonction VarType renvoie 11 quand la variable est un booléen.

Le type Objet

Ce sont des variables faisant référence à des objets. Ne me dites pas que vous avez oublié ce que c'est... Ce sont les classeurs, les feuilles, les cellules ou même l'application. On a vu dans le cours qu'avec la fonction With... End With on peut faire les faignants et ne pas tous réécrire. Et bien là, on va pouvoir mettre des objets en tant que variable. Ce type de variable occupe 4 octets. L'instruction Set permet d'attribuer une référence d'objet à la variable. Par contre, ce type de variable prend de la place et on libère donc la mémoire grâce à Nothing (cf. exemple ci-dessous).

Les exemples sont plus explicites que le texte :

```
Sub test()



' On crée une variable qui a pour type une feuille.

' De ce fait, on peut lui attribuer une valeur spécifique : une feuille d'un classeur.

Dim Ws As Worksheet



' On attribue la 1ere feuille du classeur dans la variable grâce à Set

Set Ws = Sheets(1)
```

```
' On affiche le nom de la première feuille

MsgBox Ws.Name



' On libère ensuite la mémoire

Set Ws = Nothing



End Sub
```

Ceci est réalisable avec une feuille ou un classeur ou une cellule. En combinant tout ça, on obtient ceci :

```
' On attribue des variables : c comme classeur pour Workbook.

' f comme feuille pour Worksheet.

Dim c As Workbook, f As Worksheet



' On associe à c le premier classeur

Set c = Workbooks("classeur1")



' On associe à f la feuille 1 du classeur 1

Set f = c.Worksheets("feuil1")



' On affiche la valeur de la première cellule de la première feuille,

' du premier classeur

MsgBox f.Range("A1").Value
```

On aurait pu aussi associer Range("A1") à une variable pour raccourcir le code si on avait à l'utiliser souvent. Ce code est utile et remplace ce code :

```
Workbooks("classeur1").Worksheets("feuil1").Range("A1").Value
```

Ce que nous avons fait avant peut paraitre plus long, mais très pratique par la suite, en effet, la simple lettre f remplace tout ce code :

```
Workbooks("classeur1").Worksheets("feuil1")
```

Voilà, vous connaissez la base sur les variables. Vous pensiez que c'était assez ? Eh bien pas tout à fait, c'est pour ça que l'on a décidé de refaire un chapitre sur les variables ! Ce n'est pas pour nous répéter ne vous inquiétez pas, c'est évidement pour apporter encore plus de connaissances et continuer votre route vers les beaux programmes VBA ! 

Les variables 2/2

Nous avons étudié dans le chapitre précédent ce qu'était une variable, comment la déclarer et comment l'afficher grâce à MsgBox. Enfin on a vu les différents types de variables. Vous croyez que c'est fini ? Non, non, non, il reste encore pas mal de choses à connaître sur les variables et c'est pour ça qu'il y a ce deuxième chapitre consécutif concernant les variables. A la fin de celui-ci, vous devriez connaitre le plus important sur les variables.

Les tableaux

Pour l'instant, on a déclaré que des variables individuelles, c'est-à-dire des variables une à une. Imaginez que vous ayez une liste d'invités à faire. Vous allez certainement déclarer comme ceci :

```
Dim invite1 As String, invite2 As String, invite3 As String, invite4 As String
```

Sacrée soirée si vous n'avez que 4 invités !

Ouais mais bon, créer 50 variables c'est un peu galère...
Je ne suis pas tout à fait d'accord, il suffit d'avoir les bons outils. Bon j'avoue que je ne vous ai pas aidé pour l'instant, mais on va utiliser un outil très puissant : **les tableaux**.

Comment déclarer un tableau ?

C'est comme pour les variables simples, on utilise Dim puis le nom de la variable et enfin le type de la variable.

```
Dim invite(50) As String
```

Voilà, nous avons créé un tableau de 51 cellules sur une colonne. Un tableau fonctionne comme une feuille Excel, la colonne A est notre tableau et chaque ligne est une entrée. On peut ainsi mettre en cellule A1 le nom du premier invité en A2, le deuxième, ainsi de suite. En VBA, on n'utilise pas A1, A2... mais invite(0), invite(1)...

Un tableau commence à 0 et non à 1, c'est une source d'erreurs pour les néophytes en VBA.
Dans l'exemple précédent, on a donc créé 51 variables invite numérotées de 0 à 50 et de type String. On peut donc maintenant attribuer un nom à chaque invité.

```
invite(0)="Robert"

invite(1)="Marcel"

invite(2)="André"

invite(3)="Vital"

...
```

Un tableau peut prendre tous les types de variables et il est limité à 32 767 éléments. 

Nous avons vu que les tableaux commençaient à 0. On peut modifier cette base avec la fonction **Option Base**. Comme pour la fonction Option Explicit, il faut la placer avant tout code dans un module dans la section de déclarations comme ceci :

```
Option Base 1
```

Les tableaux commenceront alors à 1. Cette base de 1 peut être unique à un tableau en utilisant le mot-clé **To**.

```
Dim mon_tableau(1 To 15) as String
```

Ce tableau comprend 15 éléments de type String.

Les tableaux multidimensionnels

Les tableaux que l'on a vus pour l'instant sont à une dimension. Pour l'image, la dimension est la colonne A de notre tableur Excel. En VBA, on peut créer des tableaux à plusieurs dimensions. Pour un tableau à deux dimensions, c'est simple, on va renseigner le nombre de colonnes puis le nombre de lignes. Le tableau ressemble alors à une feuille de calculs classique.

```
' On crée un tableau de 11 colonnes et 5 lignes

Dim mon_tableau(10, 4) As String




' On peut alors attribuer des valeurs aux différents éléments du tableau

mon_tableau(0, 0) = "Première cellule"

mon_tableau(4, 2) = "Cellule de la 5ème colonne et 3ème ligne"
```

Ce n'est pas fini. Même si ce tableau comporte alors 55 cellules, on peut le dupliquer dans une troisième dimension. Reprenons l'exemple du classeur Excel, vous avez votre feuille qui est un tableau à 2 dimensions et pouvez la dupliquer ! À chaque fois, le nombre de cellules dans le tableau augmente. Ça fonctionne toujours de la même façon :

```
Dim super_tableau(5, 3, 6) As String
```

Ce tableau comporte donc 7 feuilles de 6 colonnes et 4 lignes. Ce qui nous fait 168 cellules et éléments qui peuvent prendre une valeur.

On dit que le nombre de dimensions peut monter jusqu'à 5, mais la gestion de ces tableaux devient très difficile. De plus, plus on donne de dimensions, plus on augmente l'espace utilisé par la mémoire. De ce fait, au bout d'une certaine quantité de mémoire utilisée, le programme peut ralentir le fonctionnement de la machine. Attention donc à ne pas utiliser de l'espace inutilement.

Portée d'une variable

Si vous déclarez une variable au sein d'une procédure, seul le code au sein de cette procédure peut accéder à cette variable. La portée est locale à cette procédure. Vous aurez souvent besoin de variables qui doivent êtres utilisées par plusieurs procédures ou même par la totalité de l'application. Pour toutes ces raisons, vous pouvez déclarer une variable au niveau local, au niveau du module ou bien au niveau global.

Les variables locales

Une variable locale est utilisée dans une procédure et uniquement dans cette procédure si bien qu'une fois la procédure terminée, la variable n'est plus accessible. De ce fait, une variable appelée ma_variable peut avoir une valeur dans une première procédure et ce même nom de variable utilisé dans une autre procédure peut avoir une autre valeur. En effet, on aura créé deux variables différentes puisque dans deux procédures différentes. Par défaut, on crée des variables locales.

Les variables de niveau module

Comme son nom l'indique, la variable de niveau module permet d'y accéder partout dans un module et donc dans toutes les procédures mais pas dans le reste de l'application. Elle reste en vie durant tout le temps d'exécution de l'application et conserve leur valeur.

Pour qu'elle soit utile dans tout le module, la variable doit être déclarée avant toute procédure dans les *déclarations*.

Les variables globales

Les variables globales sont déclarées dans la section des déclarations comme pour les variables de module. Par contre, une variable globale va être précédée de Global au lieu de Dim. Les variables globales existent et conservent leur valeur pendant toute la durée de l'exécution de l'application. Cette variable globale est déclarée au début de n'importe quel module, car celle-ci est accessible dans tous les modules.

```
Global ma_variable_globale
```

Les variables statiques

Toutes les portées de variables que l'on vient de voir ont une durée de vie définie et finissent par mourir. Les variables locales sont réinitialisées quand la procédure se termine, les variables de module et globales sont réinitialisées et donc leurs valeurs sont détruites après la fin de l'exécution de l'application. Pour pallier à la destruction des variables, on peut remplacer Dim par Static pour que les valeurs soient préservées même après la fin de l'exécution de l'application ou de la procédure.

```
Static ma_variable_statique
```

Si vous ne voulez pas mettre le mot Static devant toutes les variables d'une procédure, vous pouvez ajouter Static avant la déclaration de procédure.

```
Static Sub procedure_variables_statiques
```

Conflits de noms et préséance

Une variable ne peut changer sa portée au cours de l'exécution du code. Vous pouvez cependant avoir deux variables du même nom, mais pas de la même portée. Par exemple, vous avez une variable ma_variable de type global et une autre variable ma_variable de type local dans une procédure. Dans cette procédure, on peut donc faire référence à deux variables différentes qui portent le même nom. La variable locale aura le dessus sur la variable globale. Si on veut appeler la variable globale dans cette procédure, il faut alors changer le nom d'une des variables. Cette notion de préséance est difficile à intégrer et est une source d'erreur. Le meilleur moyen de l'éviter, c'est de donner des noms différents à toutes les variables.

Les constantes

Les constantes sont en réalité des variables qui ne sont jamais modifiées. Elles contiennent des valeurs qui sont utilisées sans arrêt dans le code et fournissent des raccourcis pour ces valeurs constantes particulières.

On ne peut évidement pas modifier une constante et pour la déclarer on n'utilise pas Dim mais Const. Les constantes sont utiles dans plusieurs contextes à vous de voir dans lesquels elles vous seront utiles.

```
Const nom_chemin = "D:\Utilisateur\"
```

Il existe des constantes prédéfinies dans le modèle d'objet d'Excel que l'on étudiera plus tard dans ce cours.

Définir son type de variable

C'est vrai on peut créer nos types de variables ?

Si je vous le dis. Je ne vais quand même pas jouer avec votre cœur  . Grâce au mot clé Type, on peut créer son propre type de variable à partir des types déjà existant. La définition du type doit être avoir lieu dans la section des déclarations d'un module.

```
Type joueur

    tribu As String

    vies As Byte

    points As Integer

End Type
```

Ce code crée un nouveau type appelé joueur qui contient des informations sur la tribu, le nombre de vies et le nombre de points d'un joueur. Ce type s'ajoute automatiquement à la liste déroulante comme les autres types déjà existants. On peut l'utiliser comme les autres types. Exemple :

```
' On déclare une variable joueur

Dim bat538 As joueur



' On renseigne les différents champs du type

bat538.tribu = "Zér0"

bat538.vies = 5

bat538.points = 0



'On affiche un à un les paramètres

MsgBox bat538.tribu

MsgBox bat538.vies
```

```
MsgBox bat538.points
```
Notez que la variable bat538 créée comporte une liste déroulante qui recense les champs du type de données quand vous saisissez le nom de la variable dans le code.

Vous pouvez aussi créer un tableau à partir du nouveau type créé.

Ouf ! Enfin ! C'est fini ! On vient de terminer une grosse partie sur les variables, mais très importante puisqu'on les utilise à tout bout de champ dans les programmes. On va maintenant voir comment fonctionnent les fonctions, modules et sous-routines. Allez, hop c'est parti, on continue ! Enfin, je vous accorde une pause pour vous ressourcer...

Les conditions

Ce chapitre est essentiel à la programmation, mais pas difficile. Super non ? Enfin, il est facile à comprendre et présent dans la plupart des langages de programmation. Les conditions, c'est des choix que fait le programme en fonction de critères. Chacun est habitué à la prise de décision. Après tout, vous prenez des décisions tous les jours. Par exemple, quand vous vous réveillez, il vous faut choisir vos habits. Vous prenez cette décision en fonction de différents facteurs comme la météo ou bien votre type d'activité de la journée.

Les programmes doivent également prendre des décisions en fonction de paramètres auxquels le programme a accès. Les programmes informatiques seraient extrêmement monotones s'ils ne prenaient jamais de décision. Par exemple, si un programme tente de charger un classeur et qu'il ne trouve pas ce classeur, il faut prendre une décision pour savoir quelle conduite adopter. Est-ce que le programme doit simplement afficher un message d'erreur et stopper son exécution ou bien doit-il faire preuve de plus d'intelligence et alerter l'utilisateur que le fichier est manquant et offrir une action alternative ?

Qu'est ce qu'une condition ?

Nous venons de voir l'utilité de la prise de décision dans un programme pour qu'il puisse se débrouiller seul avec le moins d'interventions possible de l'utilisateur. Elle gère les différents évènements d'un programme et devient de suite une notion très importante.

Pour démarrer, on va expliquer ce qu'est une condition. Une condition commence toujours par un SI (if en anglais). Dans la vie courante, on peut dire : "Si je finis de manger avant 13 h, je vais regarder le journal télévisé". On peut aussi aller plus loin en disant "Sinon, j'achète le journal". Pour VBA, c'est la même chose. On a une commande conditionnelle dans VBA qui nous permet de faire des conditions : **If...Then...Else**.

Pour faire une condition, il faut un critère de comparaison. Lorsque vous faites un puzzle, vous triez en premier les pièces qui font le tour pour délimiter le puzzle et aussi parce que le critère de comparaison entre les pièces est simple : sur les pièces du tour, il y a un côté plat. Donc lorsque vous prenez une pièce en main, vous comparez les côtés de la pièce à un côté plat et vous la mettez soit dans la boite des pièces du tour soit dans les pièces qui seront retirées par la suite.

Dans VBA, ce critère de comparaison est soit une valeur, une variable ou encore du texte. On compare les données d'une cellule à notre critère de comparaison et VBA renvoie VRAI si la comparaison est juste ou FAUX. VBA exécute enfin ce que vous lui avez dit de faire en fonction de ce que renvoie la comparaison.

Pour comparer des valeurs numériques ou même du texte, on utilise des signes mathématiques. Le plus connu des signes de comparaison est égal à (=). Si les valeurs sont égales, alors fait ceci sinon fait cela. Je vous donne la liste de tous les opérateurs utilisés dans VBA pour les comparaisons :

Opérateur de comparaison	Signification
=	Égal à
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égal à
<=	Inférieur ou égal à
<>	Différent de

On peut donc avec ces opérateurs de comparaison faire toutes les comparaisons possibles entre deux valeurs.
On va alors s'entraîner à faire des comparaisons avec des données :

Comparaison
age <= 1
mot_de_passe <> password
pointure > 39
pseudo = identifiant
age >= 18
parents < 3

Alors, vous savez à quoi correspondent toutes ces comparaisons ? La réponse juste en dessous.

Comparaison	Signification (Si...)
age <= 1	La variable age est inférieure ou égale à 1
mot_de_passe <> password	Les variables mot_de_passe et password sont différentes
pointure > 39	La variable pointure est supérieure à 39
pseudo = identifiant	Les variables pseudo et identifiant sont identiques
age >= 18	La variable age est supérieure ou égale à 18
parents < 3	La variable parents est inférieure à 3

Vous voyez que l'on peut faire plein de choses avec ces comparaisons !

Créer une condition en VBA

Pour créer une condition en VBA, on a besoin de plusieurs mots-clés. Le premier c'est If, ce mot clé permet d'ouvrir la condition avec un SI. Ensuite on écrit la condition avec le critère de comparaison. Lorsqu'on a fini, on inscrit le mot-clé Then qui signifie "alors". Cela signifie que l'on va dire ce qu'il faut faire si la condition est respectée.

Le mot-clé Else nous permet d'écrire du code si la condition n'est pas respectée.

Pour terminer la condition, on écrit le mot clé End If comme pour achever une macro avec End Sub.

Un exemple pour illustrer et bien comprendre :

```
Sub test_condition()



 Dim age As Byte

 age = 20



' Si vous avez un age supérieur ou égal à 18 :



 If age >= 18 Then

  MsgBox "Vous êtes un adulte"



' Sinon :



 Else

  MsgBox "Vous êtes encore mineur"



 End If

End Sub
```

Dans un premier temps, on crée une variable à laquelle on applique une valeur (on pourra dans la suite du cours avoir une interaction avec l'utilisateur, pour l'instant changez vous-même la valeur de la variable).

Ensuite on compare la variable à l'âge de la majorité et on affiche une phrase en fonction du résultat.

On indente notre code dans une condition et aussi le reste du code pour que ce soit plus lisible. Indenter c'est créer un espace entre le bord gauche et le début de la commande pour bien voir les niveaux de la commande. Si l'on souhaite faire plusieurs conditions, on n'est pas obligé de créer un groupe de If...Then...Else à chaque fois. On peut comparer notre variable à une première valeur, si c'est vraie alors on exécute le code, sinon on compare à une autre valeur, si cette fois c'est vrai on exécute le code et ainsi de suite tant qu'il y a du code à comparer. Jusqu'à ce que vous n'ayez plus de valeur à comparer.

```
Sub test_condition()

 Dim age As Byte

 age = 20



 If age >= 18 Then

  MsgBox "Vous êtes un adulte"



 ElseIf age < 8 Then

  MsgBox "Vous êtes un surdoué pour être ici à votre âge !"



 Else

  MsgBox "Vous êtes encore mineur"



 End If

End Sub
```

Vous remarquez qu'on utilise le mot-clé **ElseIf** pour faire une deuxième comparaison et afficher du texte en conséquence. Cette ligne avec le ElseIf fonctionne comme la ligne du If.

Simplifier la comparaison multiple

Il existe une autre commande disponible en VBA pour traiter des conditions : **Select Case**. Cette commande permet de simplifier la succession de ElseIf. Si vous avez une variable et que vous vouliez accomplir différentes actions en fonction de la valeur de cette variable, vous pouvez utiliser Select Case ainsi :

```
Sub test_note()
```

```
Dim note As Byte

note = 12.4



Select Case (note)

 Case Is < 8

   MsgBox "Vous n'avez pas votre baccalauréat"



 Case Is < 10

   MsgBox "Vous devez aller aux rattrapages"



 Case Is < 12

   MsgBox "Félicitations ! Vous avez votre bac !"



 Case Is < 14

   MsgBox "Vous avez la mention Assez Bien"



 Case Is < 16

   MsgBox "Vous avez la mention Bien"



 Case Is < 18

   MsgBox "Vous avez la mention Très Bien"



 Case Else

   MsgBox "Vous avez les félicitations du jury et les miennes par la même occasion"
```

```
 End Select

End Sub
```

Ce code est beaucoup plus lisible que avec des ElseIf partout et surtout plus rapide. Vous connaissez alors une alternative pour comparer une variable dans un même bloc.

Un autre exemple pour que vous connaissiez les différentes formes de Case.

```
Select Case (note)

 Case 1

  MsgBox "La note est de 1"



 Case 2, 3

  MsgBox "La note est de 2 ou 3"



 Case 4 To 6

  MsgBox "La note est de 4, 5 ou 6"



 Case Is > 6

  MsgBox "La note est supérieure à 6"



 Case Else

  MsgBox "La note est inférieure à 1"



End Select
```

Dans cet exemple, il y a des petites nouveautés avec les deux valeurs séparées par une virgule ou les valeurs séparées par **To** qui signifie "de ... à". Le code et ce qui est dans les MsgBox suffit à comprendre la signification de chaque Case.

Commandes conditionnelles multiples

Dans les commandes précédentes, on a utilisé une seule condition de la forme " Si age >= 18 alors ... ". Vous pouvez aussi utiliser plusieurs commandes conditionnelles en utilisant un opérateur logique. Un opérateur logique, c'est par exemple un simple AND (qui signifie ET en français). Avec cet opérateur, on peut comparer notre variable age à deux critères dans la même condition.

Présentations des deux opérateurs logiques AND (ET) et OR(OU)

L'opérateur AND exige que toutes les conditions soient vraies pour exécuter la première partie du code. Si une seule des conditions est fausse, alors la procédure exécute le code situé après le Else. L'opérateur OR exige qu'une seule des conditions soit vraie pour exécuter la première partie du code.

Vous avez compris ? Pas trop n'est-ce pas. Et bien on va voir toutes les possibilités dans deux conditions avec l'opérateur **AND** et deux conditions avec l'opérateur **OR**. Pour chaque ligne, on donne ce que renvoie la condition 1 et ce que renvoie la condition 2 puis le résultat que renvoie la commande **If**. Des exemples très simples sont mentionnés pour vous aider à comprendre.

Condition 1 :	note < 10
Condition 2 :	travail < 5

	Condition 1	Condition 2	Résultat
AND	VRAI	VRAI	VRAI
	VRAI	FAUX	FAUX
	FAUX	VRAI	FAUX
	FAUX	FAUX	FAUX

	Condition 1	Condition 2	Résultat
OR	VRAI	VRAI	VRAI
	VRAI	FAUX	VRAI
	FAUX	VRAI	VRAI
	FAUX	FAUX	FAUX

On va pouvoir alors exécuter du code en fonction de deux variables. Dans un premier temps en utilisant l'opérateur AND. Si la note est inférieure à 10 et que le travail est inférieur à 5h, alors, il faut motiver l'élève à travailler plus pour avoir une meilleure note.

Il est très simple d'utiliser ces opérateurs. Il suffit d'écrire ce code :

```
' Pour l'opérateur And c'est comme ceci :

If age < 18 And sexe = "Masculin" Then

 MsgBox "Vous n'êtes encore qu'un garçon..."



Else

 MsgBox "Soit plus de 18 ans, soit vous êtes de sexe féminin"
```

```
End If



' Pour l'opérateur Or C'est la même chose



If pointure < 40 Or taille < 34

 ...

End If
```

Il faut retenir qu'avec And, toutes les conditions doivent être vraies pour exécuter le code qui suit Then. Alors que pour Or, une seule de TOUTES les conditions doit être vraie pour exécuter la première partie du code.

D'autres opérateurs logiques

Il existe d'autres opérateurs logiques pour vos conditions qui permettent d'autres comparaisons.

Not

L'opérateur Not inverse le résultat d'une condition. Si une condition est vraie, elle devient fausse avec le mot-clé Not.

```
If Not(2 = 3) Then

 MsgBox "Les valeurs ne sont pas égales"



Else

 MsgBox "Les valeurs sont égales"



End If
```

Grâce à l'opérateur Not, même si la condition est fausse, on exécute la première partie du code.

Xor

Cet opérateur logique est très semblable à Or. La différence, parce que oui il y en a une, est que pour Xor si deux conditions sont vraies, alors c'est le code de la deuxième partie qui est exécutée. VBA considère que deux conditions vraies donnent faux avec Xor. 

C'est tout ce que je peux vous dire sur cet opérateur. C'est avec l'expérience que vous serez amené à l'utiliser. Pour l'instant, vous savez qu'il existe et savez l'utiliser.

Is

L'opérateur Is permet de comparer deux objets VBA.

```
Sub test()



If Worksheets(1) Is Worksheets(1) Then

 MsgBox "Les deux feuilles sont identiques"

Else

 MsgBox "Les deux feuilles ne sont pas identiques"

End If



If Worksheets(1) Is Worksheets(2) Then

 MsgBox "Les deux feuilles sont identiques"

Else

 MsgBox "Les deux feuilles ne sont pas identiques"

End If



End Sub
```

Le premier code va afficher la première réponse alors que le second va afficher la seconde même si les deux feuilles sont vierges parce qu'elles ont un nom différent.

Like

Cet opérateur permet de comparer deux chaînes de caractères. La correspondance entre deux chaînes peut être seulement les *n* premiers caractères ou simplement au niveau de la casse (majuscule ou minuscule).

On utilise un mot-clé **Option Compare** que l'on doit écrire dans la section des déclarations. Ce mot clé peut être suivi de deux valeurs : Binary ou Text. Si vous spécifiez Binary, une lettre en majuscule et la même lettre en minuscule ne sont pas considérées comme identiques. Alors qu'avec Text, la comparaison ne fait pas de différences entre minuscule et majuscule. Si Option Compare est utilisé seul, c'est Binary par défaut qui est utilisé.

Dan cet exemple on utilise le mode Binary :

```
Option Compare Binary
```

```
Sub test()



If "Site du Zéro" = "site du zéro" Then

 MsgBox "Les chaînes sont exactement les mêmes"



Else

 MsgBox "Il y a des différences entre les chaînes de caractères"



End If



End Sub
```

Dans ce code, les chaînes ne sont pas identiques puisque des majuscules sont dans la première chaîne et absentes dans la seconde. Si on exécute le même code en spécifiant Text à la place de Binary, c'est le premier message qui sera renvoyé.

Si on veut comparer que quelques caractères, on peut utiliser des caractères joker. Le symbole ? remplace un seul caractère, mais n'importe lequel, le symbole * remplace quant à lui une chaîne de caractère. Je vous présente un tableau avec les caractères de comparaison et leur signification.

Caractères	Signification
?	Remplace un seul caractère
*	Remplace zéro ou plusieurs caractères
#	Remplace un chiffre (de 0 à 9)
[*Liste_de_caractères*]	Remplace un seul caractère de la liste
[*!Liste_de_caractères*]	Un seul caractère absent de la liste.

Comment utiliser ces caractères ? Pour les trois premiers, c'est assez simple, il suffit de remplacer les caractères à remplacer par ?, * ou #. Faites des tests pour bien comprendre (des exemples sont présentés juste après).

Pour ce qui est des listes de caractères, on peut faire quelques précisions.

Par exemple, vous écrivez cette liste : [abcdef]. Cela signifie que le caractère de la première chaîne doit être

soit un a, soit un b, soit un c, soit un d, soit un e, soit un f pour que la commande renvoie Vrai. Si le caractère de la première chaîne n'est pas une de ces lettres, alors la commande renvoie Faux.

Si je veux que ce soit toutes les lettres de l'alphabet, on écrit tout l'alphabet ?
Rappel : les programmeurs sont des feignants. Il existe alors une simplification ! Si vous voulez les lettres de a à f comme dans l'exemple précédant, il suffit de mettre un trait d'union (-) entre la première et la dernière lettre : [a-f], ceci correspond à [abcedf].

De la même façon, [a-z] équivaut à [abcdefghijklmnopqrstuvwxyz] et [0-9] correspond à [0123456789].

Nous allons mettre en place une série d'exemples pour bien comprendre et utiliser. Il faut alors initialiser Option Compare à Text.

```
Option Compare Text



Sub test_comparaison()



If "site du zéro" Like "site du ?éro" Then

' Cette ligne renvoie True



If "site du zéro" Like "site du *" Then

' Cette ligne renvoie True



If "Site du zéro" Like "site ?u zero" Then

' Cette ligne renvoie False non pas parce que le "S" est en majuscule puis minuscule

' ni à cause du ?, mais parce que dans la première chaîne, il y a un "é"

' et dans la seconde, il n'y a pas d'accent.



If "Site du 0" Like "site du #" Then

' Cette ligne renvoie True car 0 est un chiffre.
```

```
If "Site du 0" Like "site du [012345]" Then

' Cette ligne renvoie True car le caractère de la première chaîne est présent dans

' la liste entre crochets.



If "Site du 0" Like "site du [5-9]" Then

' Cette ligne renvoie False car 0 n'est pas compris entre 5 et 9 compris.



If "Site du 0" Like "site du [!5-9]" Then

' Cette ligne renvoie True car 0 n'est pas compris entre 5 et 9 compris.

' C'est l'inverse de l'exemple précédant.



If "Site du zéro ?" Like "site du zéro [?]" Then

' Cette ligne renvoie True car le caractère à la fin est bien un point d'interrogation.

' N'oubliez pas les crochets, sinon, n'importe quel caractère peut être à la place

' du point d'interrogation dans la première chaîne.



End Sub
```

J'espère que ces exemples vous ont été utiles pour comprendre l'opérateur Like. Sinon, n'hésitez pas à relire et faire des tests vous-mêmes.

Après avoir développé un peu d'intelligence artificielle dans notre programme, il est temps de voir de nouvelles notions qui permettront d'engager un "dialogue" plutôt humain entre l'utilisateur et le programme. Nous allons donc attaquer **les boucles**, qui exécutent un morceau de code plusieurs fois de suite tant qu'une condition est vraie. Comme quoi, tout est lié ! 

Les boucles

Dans ce chapitre, nous allons reprendre la notion fondamentale du chapitre précédent : les conditions. Une boucle permet de répéter une action tant qu'une condition renvoie Vrai. De ce fait, on pourra faire des actions répétitives très facilement.

Prenons un exemple, vous avez un tableau et vous voulez remplir toutes les entrées par les lettres de l'alphabet. Au lieu de prendre une ligne pour chaque entrée, on va créer une boucle qui permet d'affecter à chaque entrée une lettre de

l'alphabet.

Ce que je viens de vous proposer, c'est une boucle simple où il y aura qu'une instruction, mais on pourra complexifier ces boucles très facilement par la suite.

Les boucles, comme les conditions, on les utilise dans la vie de tous les jours. Par exemple, vous êtes en train de marcher. L'instruction à l'intérieur de la boucle est : "faire un pas", la condition peut être multiple du type : "tant que je ne suis pas arrivé et qu'il n'y a pas de trou alors ...". On boucle cette instruction pour que nous puissions enchainer les pas et arriver à destination sans tomber dans un trou. S'il fallait écrire une instruction à chaque pas, le code serait très long !

La boucle simple

Dans cette première partie, nous allons étudier le code d'une boucle simple. Pour cela il faut d'abord savoir ce que fait une boucle et quel code elle remplace. Imaginons un code très simple et basique pour afficher les nombres de 1 à 10 dans une boite de dialogue en utilisant MsgBox. Voici la première solution que vous êtes en mesure de coder :

```
Sub boucle()



 MsgBox "1"

 MsgBox "2"

 MsgBox "3"

 MsgBox "4"

 MsgBox "5"

 MsgBox "6"

 MsgBox "7"

 MsgBox "8"

 MsgBox "9"

 MsgBox "10"



End Sub
```

Et encore, heureusement que je vous ai demandé jusqu'à 10  Imaginons que je vous demande jusqu'à 100. Il vous faut ajouter 90 lignes de code, ça pompe de l'énergie au programmeur (qui est un être de nature fainéant), ça ralentit le programme et c'est très inefficace. Venons-en à la notion clé de ce chapitre : la boucle. On utilise la boucle que je vais appeler **For..Next**. Voici à quoi peut être réduit le code précédent :

```
For nombre = 1 to 10



 MsgBox nombre



Next nombre
```

La boîte de message s'affiche 10 fois de 1 à 10.

Comment fonctionne ce code ?

Tout d'abord, on utilise le mot-clé **For** qui annonce le début de la boucle. Ensuite on utilise le nom d'une variable : ici **nombre** mais on peut mettre ce que l'on veut tant que l'on garde ce nom de variable dans toute la boucle. On dit où commence la boucle, ici 1 mais on aurait très bien pu afficher à partir de 5 seulement (tester par vous même). Le mot-clé suivant est **To** pour dire "jusqu'à" puis la valeur de la dernière valeur à afficher. La première ligne est maintenant terminée.

Sur la seconde ligne et les suivantes, on écrit ce que l'on veut, c'est le code à exécuter. On n'est pas obligé d'utiliser la variable nombre dans ce code, si on veut juste afficher le texte "Hip" 3 fois, on crée une boucle qui affiche le texte sans se préoccuper de la variable.

Enfin quand vous avez fini votre boucle, sur la dernière ligne, il faut dire au programme de passer à la valeur suivante par le mot-clé **Next** suivi du nom de la variable. Ainsi, le programme recommence au début avec la valeur suivante jusqu'à ce que la valeur atteigne la valeur plafond (ici 10).

Le mot-clé Step

En anglais, *step* signifie *pas*. C'est le pas à plusieurs significations : il y a le pas lorsque vous marchez, mais la signification qui nous intéresse ici est le pas comme écart entre deux valeurs. Dans notre boucle, par défaut le pas est de 1, c'est-à-dire que la variable est incrémentée (augmentée) de 1 à chaque tour de boucle. Grâce au mot-clé **Step**, on peut modifier ce pas. On va alors afficher que les chiffres pairs de 0 à 10.

```
For nombre = 0 to 10 Step 2



 MsgBox nombre



Next nombre
```

Ce code affiche successivement 0, 2, 4, 6, 8, 10 parce que le pas est de 2, vous pouvez aussi mettre un nombre à virgule pour aller de 0 à 1 par exemple en ajoutant à chaque fois 0,1.

Dans VBA, une virgule est indiquée par un point (.) et non par une virgule (,).
Vous pouvez alors avec ce mot-clé incrémenter à votre guise.

Encore plus fort ! Step fonctionne à l'envers ! On peut partir de 10 et arriver à 0 en affectant à Step la valeur de -1. On peut ainsi créer un compte à rebours comme ceci :

```
Sub rebours()



 For n = 5 To 1 Step -1



  MsgBox n



 Next n



 MsgBox "Boom !"



End Sub
```

Ce code effectue le compte à rebours à partir de 5 jusqu'à 1 avant que ça ne pète !

Cette boucle For..Next est très pratique pour balayer les numéros de ligne ou colonne.

Un dernier exemple avec l'affectation à chaque entrée d'un tableau une lettre de l'alphabet :

```
Sub alphabet()



 Dim lettre(25) as String



 For n = 0 To 25

  lettre(n) = Chr(n+65)

 Next n



 For n = 0 To 25

  MsgBox lettre(n)

 Next n
```

```
End Sub
```

Dans un premier temps, on crée un tableau de 26 entrées (nombre de lettres de notre alphabet). Dans une première boucle, on affecte à chaque entrée une lettre de l'alphabet, dans la seconde boucle on affiche ces lettres.

Euh, c'est quoi Chr(n+65) ?

Chr est une fonction de VBA qui permet de renvoyer la lettre qui correspond à un code numérique. Ce code est le ASCII (*American Standard Code of Information Interchange* ou Code standard américain pour les échanges d'information). Ainsi grâce à ce code chaque nombre correspond à une lettre et par exemple A = 65, B = 66 et ainsi de suite. Comme notre code commence à 0, il faut ajouter 65 pour avoir la lettre A.
Quand *n* est égal à 1, on ajoute 65 pour obtenir 66 et la lettre B. La fonction Chr permet de convertir ce nombre en lettre et on affecte cette valeur à l'entrée du tableau correspondante.

Si le sujet vous passionne, vous trouverez ici la table ASCII complète 

Les boucles imbriquées

On peut imbriquer les boucles pour par exemple parcourir les cellules d'une feuille de calculs. On va alors afficher les coordonnées de chaque cellule d'un tableau de 5 lignes et 5 colonnes. Voici le code, une petite explication juste après :

```
Sub test_tableau()



 For colonne = 1 To 5

  For ligne = 1 To 5

   MsgBox Chr(colonne+64) & ligne

  Next ligne

 Next colonne



End Sub
```

Dans la première boucle, on utilise la variable *colonne* qui va représenter la colonne. Dans le premier tour de boucle, on utilise la valeur 1 pour *colonne*. On ajoute 64 à la valeur de *colonne* pour obtenir la valeur dans le code ASCII et affiche la lettre correspondant à la colonne.

Dans la deuxième boucle, on boucle le numéro de ligne grâce à la variable *ligne*. Ainsi, on va afficher tour à tour toutes les cellules de la première colonne, la colonne A. On obtient donc A1, A2, A3, A4 ,A5. Une fois que la boucle des lignes est terminée, la boucle sur les lignes est terminée et le code continue à s'exécuter. Il arrive à la ligne *Next colonne*. De ce fait, le code reprend au début de la boucle, avec la colonne B et refait la boucle concernant les lignes pour afficher toutes les cellules de la colonne B et ainsi de suite. Il y a donc 25 boites de dialogue qui s'affichent.

Cet exemple est terminé, j'espère qu'il vous a servi et que vous avez compris. Faites des tests chez vous, c'est comme ça que l'on apprend ! Passons maintenant à une autre boucle.

La boucle For Each

La boucle **For Each** est très semblable à la boucle que nous venons de voir. Elle permet de se déplacer dans un tableau très facilement ou de parcourir une classe d'objets.

On n'utilise pas d'indice dans cette boucle parce qu'elle parcourt automatiquement tous les éléments de la classe (aussi appelée collection). Cela est très pratique si vous avez besoin de rechercher un objet particulier (avec une condition à l'intérieur de la boucle) au sein de la collection pour le supprimer. L'objet supprimé va décaler les objets vers le haut et il y aura donc moins d'objets à parcourir. Avec cette boucle, tous les objets sont parcourus sans erreur, avec la boucle For..Next, il y aura une erreur à la fin puisqu'il aura un objet non créé.

Prenons un exemple pour bien comprendre. Nous avons 10 feuilles dans un classeur, soit une collection de 10 feuilles.

On cherche une feuille que l'on va supprimer. Avec la boucle For Each, on parcourt toute la collection, on analyse toutes les feuilles et supprime celle que l'on veut. Une fois toute la collection parcourue, la boucle s'arrête. En revanche, avec la boucle For..Next, on va mettre un indice qui va de 1 à 10 (puisqu'on a 10 feuilles). On supprime la feuille que l'on a trouvée, de ce fait, il n'en reste plus que 9 et les restantes se décalent pour que la collection ne soit pas de "trou". Dans ce cas, la feuille qui était en 10 se retrouve en 9 et il n'y a plus de feuille 10. La boucle va continuer de tourner jusqu'à ce que l'indice atteigne la valeur de 10. A ce moment, la boucle ne va pas trouver de feuille puisqu'elles ont été décalées et il va y avoir une erreur. On va alors utiliser la boucle For Each pour parcourir les collections. 

Pour connaitre la syntaxe de cette boucle, on va afficher dans une boite de dialogue le nom de chaque feuille de calculs. Voici le code suivi d'explications :

```
Sub afficher_noms()



 Dim une_feuille As Worksheet



 For Each une_feuille In Worksheets

   MsgBox une_feuille.Name

 Next une_feuille



End Sub
```

Dans un premier temps, on créé une variable une_feuille de type Worksheet (objet feuille). On utilise la boucle For Each pour parcourir chaque feuille de la collection Worksheets (collection de feuilles d'un classeur). La ligne signifie donc : pour chaque feuille de la collection de feuilles...

On exécute alors une ligne de commande en affichant le nom de la feuille grâce à la propriété **Name** dans une boite de dialogue. Une fois que c'est fait, on passe à la feuille suivante grâce au mot-clé Next.

Comme dans la boucle For..Next, il faut utiliser le même nom de variable (ou d'indice) pour que la boucle tourne correctement.

La boucle Do Until

Encore une nouvelle boucle qui fonctionne différemment de la boucle For..Next mais qui a la même logique. Elle va continuer de tourner tant qu'une condition est respectée. Je vous présente la syntaxe grâce à un exemple et une explication ensuite :

```
Sub test_boucle()



 ma_variable = 0



 Do Until ma_variable = 10



  ma_variable = ma_variable + 1

  MsgBox ma_variable



 Loop



End Sub
```

On initialise une variable à 0. Ensuite, le mot-clé **Do Until** est utilisé. Il signifie "jusqu'à", c'est à dire, jusqu'à ce que ma_variable soit égale à 10. Dans le code, on incrémente manuellement la variable puis on affiche sa valeur. Enfin, on utilise un nouveau mot-clé pour signaler la fin de la boucle : **Loop** (qui signifie boucler en anglais).

Attention à ne pas faire de boucle infinie, c'est-à-dire une boucle où la variable n'atteint jamais la valeur plafond ! Sinon, une boite de dialogue apparaît à chaque fois que vous cliquez sur Ok. On ne peut mettre fin au code.

La boucle While.. Wend

Les boucles, ce n'est pas encore fini ! En voici une autre ici que l'on appelle **While..Wend**. Cette boucle continue de boucler tant que la condition spécifiée est vraie. La boucle s'arrête dès que la condition est fausse. Voici un exemple simple, très semblable à celui de la boucle Do Until.

```
Sub test_while()



 ma_variable = 0



 While ma_variable < 12

  ma_variable = ma_variable + 1

  MsgBox ma_variable

 Wend



End Sub
```

Comme pour la boucle précédente, on initialise la variable à 0. Ensuite on utilise le mot-clé **While** qui signifie "tant que" suivi d'une condition, ici inférieur à 12. C'est-à-dire que tant que la variable est inférieure à 12 on entre dans la boucle, sinon on en sort et on poursuit le code. Dans la suite du code, on incrémente la variable de 1 en 1 (le pas est de 1) et on affiche à chaque fois la valeur. Quand la boucle doit remonter on utilise le mot-clé **Wend**(qui signifie s'acheminer) comme on utilise Loop dans la boucle Do Until.

Sortie anticipée d'une boucle

Dans certaines circonstances, il est souhaitable que la procédure sorte de la boucle plus tôt que prévu. Si, par exemple, vous effectuez une recherche de chaîne de caractères dans les éléments d'un tableau, il n'y a pas de raison de continuer à balayer les autres éléments du tableau une fois que vous avez trouvé ce que vous recherchez. Si vous recherchez dans un tableau de plusieurs milliers d'éléments, vous pouvez gaspiller un temps précieux si vous ne sortez pas de la boucle alors que vous avez trouvé l'objet de votre recherche. Dans le cas d'une boucle For..Next, la valeur de l'indice est aussi préservée ce qui signifie que vous pouvez l'utiliser pour localiser l'endroit où votre condition était correcte. Voici un exemple :

```
Sub test_sortie()



 For variable = 1 To 100

  If variable = 50 Then

    Exit For

  End If

 Next variable
```

```
 MsgBox variable



End Sub
```

Le mot-clé pour sortir d'une boucle, c'est **Exit For** dans une boucle For..Next ou For Each. Dans le cas d'une boucle Do Until, le mot-clé pour en sortir est **Exit Do**. Vous aurez noté que dans le cas d'une boucle For..Next, la valeur de l'indice (de la variable) est préservée. Si les boucles sont imbriquées, votre code ne sort que de la boucle dans laquelle il se trouve. Il ne sortira pas de la boucle extérieure à moins que vous n'y placiez une autre instruction Exit.

Dans l'exemple précédant, on a donc une variable qui commence à 1 et va jusqu'à 100. Dans cette boucle une instruction conditionnelle (If) est utilisée pour comparer la valeur de l'indice et celui que l'on cherche. Une fois la valeur atteinte on entre dans l'instruction conditionnelle et exécute le code : la sortie. Tant que la valeur de l'indice ne respecte pas la condition, on ne prend pas en compte l'instruction qui la suit (la sortie) et on continu à boucler. Enfin, lorsque la valeur est trouvée, on sort de la boucle et affiche cette valeur.

Exercice : un problème du projet Euler sur les dates

Le Projet Euler est un site qui propose depuis plus de dix ans des problèmes de maths à résoudre à l'aide de l'outil informatique. En général, les données à manipuler sont des nombres gigantesques et il faut donc réfléchir à des algorithmes efficaces pour avoir un résultat en une minute maximum.

Il existe deux façons de résoudre un tel problème : le brute-force ou une méthode plus réfléchie et plus élégante. Le brute-force consiste à tester toutes les solutions possibles jusqu'à trouver la bonne. Il est évident que dès que les nombres dépassent les limites du raisonnable (ce qui est généralement le cas dans le Projet Euler), cette méthode est à proscrire car les temps de calculs sur un ordinateur comme le votre ou le mien sont trop longs.

Le choix de la technologie est libre et il se trouve qu'un exercice se prête bien au VBA mais aussi à l'utilisation de la fonction NBVAL() de Excel.

Nous allons résoudre le problème 19. Je vous invite à prendre connaissance des informations données dans l'énoncé, ici, je vais simplement recopier la question :

Citation : Problème 19 - Projet Euler

How many Sundays fell on the first of the month during the twentieth century (1 Jan 1901 to 31 Dec 2000)?

Que l'on peut traduire par :

Combien de premiers du mois sont tombés un dimanche entre le premier janvier 1901 et le 31 décembre 2000 ?

La méthode brute-force ici est une solution acceptable. En effet, Excel vous renverra la solution très rapidement (moins d'une minute) donc les conditions de participation au Projet Euler sont respectées. Vous pouvez néanmoins, si vous être matheux sur les bords, vous servir de l'énoncé pour résoudre le problème sur papier ou, si vous êtes moins maso, une solution informatique élégante.

Bref, vous l'aurez compris, les boucles vont nous servir à sauter de jours en jours, jusqu'à tomber sur l'année 2000, qui met fin au calcul et renvoie le résultat  .

Quelques éléments sur les dates en VBA

Il existe un type Date tout prêt qui nous permet de manipuler des dates selon un format particulier. Voyons un exemple que vous devriez comprendre sans problème :

```
Dim maDate As Date


maDate = "12/03/2000"
```

Les opérations de comparaison que vous connaissez fonctionnent très bien sur les dates (on peut donc savoir quelle date est plus vieille qu'une autre avec nos opérateurs habituels : >, <, etc.).

Des fonctions

Nous allons voir maintenant quelques fonctions de VBA pour analyser une variable de type Date.

Il faut d'abord savoir que dans Excel, tous les jours de la semaine et les mois sont numérotés. La seule particularité est que le jour de la semaine numéro 1 est un dimanche.

Pour obtenir le numéro du jour de la semaine, il nous suffit d'utiliser la fonction WeekDay(), qui attend en paramètre une variable de type Date. Cette fonction vous renverra un entier, qu'il vous suffit de comparer au tableau suivant pour connaître le jour.

Retour de WeekDay	Correspondance
1	Dimanche
2	Lundi
3	Mardi
4	Mercredi
5	Jeudi
6	Vendredi
7	Samedi

Pour connaître le jour du mois, il suffit d'utiliser la fonction Day(), qui attend en paramètre une variable de type Date, évidement.

Ainsi, avec la déclaration ci-dessus, `Date(maDate)` renverra 12.

A vous de jouer !

Entre les boucles et ces quelques éléments ci-dessus, vous avez tout pour réussir votre (premier, pour certains) exercice du Projet Euler !

Je vous recommande d'afficher chaque résultat qui correspond dans une cellule, chaque résultat les uns en dessous des autres dans une colonne (la colonne A par exemple). La fonction NBVAL() de Excel, dont vous trouverez le guide d'utilisation à la fin de cette sous-partie de l'annexe des fonctions, va vous permettre de compter toutes les cellules non vides dans votre plage et donc d'obtenir la solution du problème.

Correction

Je vous propose une correction. Nous allons d'abord voir le code VBA puis ensuite la partie sur le tableur.

```
Sub probleme19 ()



Dim dateFin As Date, dateActuelle As Date



dateActuelle = "01/01/1901" ' on commence au 01/01/1901

dateFin = "31/12/2000"



Range("A1").Select 'on sélectionne la première cellule pour y afficher le premier résultat



While dateActuelle <= dateFin

  If WeekDay(dateActuelle) = 1 And Day(dateActuelle) = 1 Then



    ActiveCell.Value = dateActuelle ' on écrit le résultat dans la cellule active

    ActiveCell.Offset(1, 0).Select

    dateActuelle = dateActuelle + 1 ' on passe au jour suivant
```

```
  Else



    dateActuelle = dateActuelle + 1



  End If



Wend



End Sub
```

Retournez voir votre feuille de calculs, votre colonne A est remplie de dates  . Dans une cellule d'une autre colonne, il ne vous reste plus qu'à rentrer la formule : = NBVAL("A1:A2000") (2000 pour être large, mais on peut aussi tester toute la colonne) pour avoir votre solution !

Naturellement, celle-ci ne sera pas divulguée ici, ce n'est pas du jeu sinon !

Critique

Cette méthode - sous Excel - est rapide et fonctionne. Cependant, si on y réfléchit, l'algorithme brute-force utilisé est assez mauvais.

En effet, dans la vie courante, si on devait appliquer la méthode brute-force avec tous les calendriers de 1901 à 2000, on regarderait uniquement les premiers du mois, et non tous les jours. On peut donc s'intéresser, dans le code VBA, aux premiers du mois et non à tous les jours.

Si cette amélioration vous intéresse pour continuer à vous exercer, à vos tableurs !

C'est déjà terminé pour ce chapitre. Vous savez maintenant faire des boucles pour répéter des actions simples qui s'exécutent selon une condition. Nous allons voir maintenant une autre manière de ne pas répéter les actions répétitives mais d'une autre manière moins monotone : les fonctions et sous-routines. Rendez-vous au chapitre suivant !

Modules, fonctions et sous-routines

Nous avons commencé à coder avec les variables et la programmation orientée objet. Maintenant on va continuer pour toujours améliorer nos programmes et permettre d'avoir une application digne de ce nom. Les modules, fonctions et sous-routines sont indispensables aux applications. Elles permettent d'alléger un programme et de le rendre plus rapide. Ne vous fiez pas aux apparences, l'image illustre juste les fonctions, ce n'est pas si compliqué  .

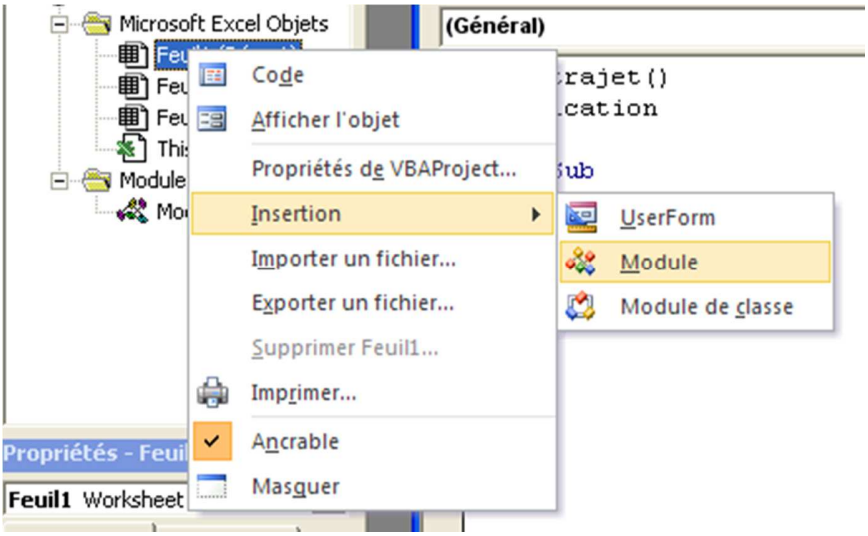
C'est en faisant qu'on apprend et l'exemple est plus explicatif que du texte non ? Alors nous allons arrêter l'introduction et passer au vif du sujet !

(Re)présentation

Les modules

Nous allons commencer en présentant ce qu'est un module. Et si je vous dis que vous le savez déjà, vous me croyez ? Il va bien falloir Un module représente la feuille dans lequel vous écrivez votre code depuis le début. Ils sont rattachés à des classeurs et le code de ces modules peut ainsi agir sur tout le classeur.

Vous pouvez aussi écrire du code qui est rattaché à une feuille ou au classeur en double cliquant sur les feuilles à gauche dans l'explorateur ou sur ThisWorkbook (qui est le classeur). On reviendra sur ce type de code plus tard.



Vous avez donc deux dossiers, les objets du classeur : Microsoft Excel Objets et les modules dans le classeur Modules. Pour ajouter un module, il suffit de faire un clic droit dans l'explorateur, aller sur insertion et cliquer sur Module.

Vous avez donc un module qui s'ajoute à l'explorateur et qui s'appelle "Module1". C'est dans celui-ci qu'on va écrire le code. Pour exécuter ce code, on a vu que l'on peut utiliser la touche F5. Mais l'utilisateur de votre application ne va pas s'amuser à ouvrir Visual Basic. On va alors affecter les macros à des Contrôles. On en a déjà parlé dans le chapitre sur la POO.

Les procédures codées peuvent aussi être exécutées en faisant appel à elle par le biais d'une autre procédure. Le détail de cette technique va bientôt venir. Les fonctions créées dans VBA peuvent aussi être appelées dans les cellules directement comme on appelle la fonction SOMME. Ça c'est top, vous aller pouvoir créer vos propres fonctions !

Différences entre sous-routine et fonctions

Vous savez qu'il y a deux types de procédures ? Non ? Et bien je vous le dis. Il existe d'un coté les sous-routines, de l'autre les fonctions. Elles sont très semblables mais en fait différentes.

Une sous-routine permet d'exécuter du code mais ne renvoie aucune valeur alors que la fonction renvoie une valeur. On a déjà utilisé des fonctions dans ce cours. C'est le cas de IsNumeric par exemple. Beaucoup de fonctions sont déjà existantes en VBA. Une grande partie des fonctions Excel sont dans VBA, il s'agira donc de les utiliser et non de les coder. Une sous routine se déclare à l'aide Sub et se termine par End Sub. Une fonction se déclare avec le mot clé Function et se termine par End Function. Si vous déclarez une fonction ou sous-routine, VBA inscrit la ligne de fin de code automatiquement, donc pas de soucis pour ça.

Le mot clé Function s'écrie avec un "u" et non un "o". C'est le mot fonction en anglais.

```
' On déclare une sous-routine

Sub ma_routine()



End Sub



'On déclare une fonction

Function ma_fonction()



End Function
```

Les fonction doivent être appelées via une variable pour qu'il y ait une valeur de retour.

```
maintenant = Now()
```

Ce code permet de mettre la date du jour dans une variable appelée maintenant.

Les deux types de procédures acceptent les paramètres. Un paramètre est une valeur que l'on envoie à une fonction ou sous-routine pour qu'elle effectue des calculs par exemple à partir de ces paramètres. Nous allons illustrer ça dans la section suivante avec un exemple.

La différence entre fonction et sous-routine est assez subtile mais doit bien être comprise pour que vous n'essayer pas de retourner une valeur avec une sous-routine par exemple.

Notre première fonction

Comment la créer ?

Nous allons maintenant créer notre première fonction. Elle sera toute simple mais c'est pour que vous compreniez le principe.

La fonction que l'on va créer existe déjà mais on va essayer de la coder quand même. Il s'agit de la fonction *Multiplier* appelée **PRODUIT** dans Excel. On va envoyer deux nombres à la fonction qui se charge de les multiplier et nous renvoie le résultat.

Je vous donne la réponse directement :

```
Function multiplier(nombre1, nombre2)



 multiplier = nombre1*nombre2



End Function
```

C'est aussi simple que ça  . Une petite description s'impose. Pour commencer, on déclare la fonction à l'aide du mot-clé Function. On donne un nom à la fonction, ici multiplier puis entre parenthèses les paramètres que prend la fonction. On multiplie deux nombres entre eux ici. Le premier paramètre est le premier nombre (nombre1) et le second paramètre est le second nombre (nombre2). Chaque paramètre est séparé par une virgule.

Ensuite on note ce qu'effectue la fonction, ici on multiplie le nombre1 et le nombre2 à l'aide de l'opérateur *. Enfin on ferme la fonction avec End Function.

Comment l'utiliser ?

Une fois votre fonction créée, elle apparait dans la liste déroulante de droite au dessus de la zone de code. Elle peut alors être utilisée dans VBA mais aussi dans Excel.

Dans Excel

Retournez dans un tableau Excel et saisissez dans une cellule :

=multiplier(3;4)

Dans une cellule, le séparateur de paramètre est un point-virgule et non une simple virgule. Une fois que vous appuyez sur Entrée, le résultat (12) s'affiche. Elle s'utilise alors comme les autres fonctions.

Dans VBA

Vous allez créer une sous-routine classique avec le nom que vous voulez. J'ai choisi par soucis d'originalité de l'appeler **test()**. On va alors appeler la fonction, lui transmettre les paramètres et afficher le résultat dans une boite de dialogue. Vous pouvez y arriver seuls mais le code suit quand même pour une correction.

```
Sub test()

 Dim variable1 As Byte, variable2 As Byte

 variable1 = 3

 variable2 = 5



 resultat = multiplier(variable1, variable2)



 MsgBox resultat

End Sub
```

Notez que lorsque vous saisissez le mot multiplier et ouvrez la parenthèse, VBA affiche automatiquement le nom des paramètres attendus. En insérant la fonction dans le code, vous forcez un appel à cette fonction en utilisant les paramètres **3** et **5** qui remplacent **nombre1** et **nombre2** dans votre fonction. Le résultat est retourné dans la variable **resultat**.

Lancez votre macro à l'aide de la touche F5. Que ce passe-t-il ? Le résultat 15 s'affiche dans la boite de dialogue.

L'intérêt d'une sous-routine

Nous avons vu qu'une sous-routine est différente d'une fonction. En ce sens, elle ne retourne rien du tout directement et elle ne peut donc pas être utilisée directement dans une feuille de calculs à la manière d'une fonction. Une sous-routine est un bloc de code que vous allez appeler d'où vous voulez dans votre programme, plusieurs fois peut-être pour vous éviter d'écrire plusieurs fois la même chose. On peut, comme pour une fonction, envoyer des paramètres mais ils seront utilisés dans la sous-routine en interne, dans le code lui-même. Aucune donnée ne peut être retournée.

Créer une sous-routine

On va dans un premier temps créer une sous-routine pour afficher le prénom de l'utilisateur.

```
Sub affichage_prenom(prenom)



 MsgBox "Votre prénom est : " & prenom



End Sub
```

Notez que cette sous-routine possède un paramètre pour une variable appelée prenom. Cela est dû au fait que vous allez appeler la sous-routine à partir d'une autre procédure et lui passer une variable.

Une ligne est tracée sous la sous-routine pour préciser où elle se termine. Elle s'ajoute alors à la liste déroulante en haut à droite.

Utiliser la sous-routine

On va maintenant utiliser cette sous-routine dans une procédure. Ce sont des procédures simples pour bien comprendre, vous pourrez par la suite complexifier vos sous-routines et procédures.

La sous-routine peut être appelée comme une fonction en affichant son nom puis les paramètres si elle en prend entre parenthèses. Mais on peut aussi utiliser le mot-clé **Call**.

```
' Ces deux lignes font la même chose

 Call ma_sous_routine(parametre)

 ma_sous_routine(parametre)
```

Dans la suite du cours, je n'utiliserais pas ce mot-clé, il permet de voir que c'est une sous-routine et qu'elle ne renvoie rien. Personnellement, je ne l'utilise pas.

```
Sub test()


```

```
 ' On crée une variable prénom et on lui attribue une valeur

 Dim prenom As String

 prenom = "Baptiste"



 ' On utilise notre sous-routine pour afficher le prénom

 affichage_prenom (prenom)



 ' On crée une variable âge ayant pour valeur 20

 Dim age As Byte

 age = 20



 ' On réutilise la sous-routine pour montrer que malgré le code entre les deux,

 ' la sous-routine continue de faire la même chose.

 affichage_prenom (prenom)



End Sub
```

A noter que j'utilise toujours la variable prenom dans mon code mais ce n'est pas obligatoire. La variable envoyée à la sous-routine ne doit pas forcément être la même que le nom de la variable utilisée dans la sous-routine. on peut très bien faire ça :

```
' On utilise toujours la même sous-routine avec la variable prenom



Sub test()



 Dim pseudo As String

 pseudo = "Baptiste"

```

```
 affichage_prenom (pseudo)



 Dim age As Byte

 age = 20



 affichage_prenom (pseudo)



End Sub
```

Vous voyez bien que j'envoie la variable pseudo à la sous-routine mais le code fonctionne de la même façon et affiche la même chose.

Les sous-routines permettent de décomposer un gros code en plusieurs portions. De cette façon, il est plus facile de résoudre un problème en prenant chaque morceaux de code les uns après les autres.

C'est un premier pas vers des programmes lisibles.

Publique ou privée ?

VBA vous permet de définir vos fonctions ou sous-routines privées ou publiques à l'aide des mots-clés **Private** ou **Public**. Voici un exemple :

```
Private Sub sous_routine_privee()


End Sub
```

Euh mes fonctions et sous-routines que j'ai créées, elles sont publiques ou privées ?
Jusque là, vous n'avez créé que des fonctions et sous-routines publiques. Par défaut, VBA les créées en publiques. Le fait qu'une fonction ou sous-routine soit publique permet à cette procédure d'être accessible dans tous les modules de l'application. De plus, elles feront partie de la liste des macros que l'on peut attribuer sur une feuille de calcul.

Vous vous doutez que pour les procédures privées, il suffit de mettre le mot-clé **Private**devant la déclaration de fonction ou sous-routine. Ainsi, vous allez pouvoir avoir des procédures qui ont le même nom dans des modules différents et qui exécutent un code différent. Ces fonctions privées ne peuvent être utilisées par un utilisateur dans la feuille de calcul.

C'est le même système que pour les variables locales ou globales.

Un peu plus loin...

Nous allons dans ce dernier point du chapitre aller plus loin en étudiant les types de données des arguments (ou paramètres) des fonctions et sous-routines.

Lorsque vous déclarez vos procédures avec des paramètres, par défaut, il sont de types Variant. On peut effectivement attribuer un autre type de données à ces paramètres (les types de données ont déjà été étudiés).

```
Function afficher_resultat(resultat As Integer)
```

Il y a un avantage à déclarer des types de données : vous introduisez une discipline dans votre code car la procédure recherche un certain type d'informations. Si vous ne spécifiez pas de type et utilisez par conséquent le type par défaut, Variant, alors votre procédure acceptera n'importe quoi, que ce soit un nombre ou une chaîne de caractères. Cela peut avoir des conséquences fâcheuses dans votre procédure si vous attendez une chaîne de caractères et recevez un nombre ou inversement. Si vous spécifiez que le paramètre est une chaîne de caractères, une erreur se produira si ce n'est pas une chaîne de caractères qui est passée en paramètre.

Les arguments optionnels

Vous pouvez rendre certains arguments optionnels en utilisant le mot-clé **Optional**.

```
Function test(parametre1 As Byte, Optional parametre2 As Byte)
```

Cette fonction demande un paramètre obligatoire de type **Byte** qui est parametre1 et un paramètre optionnel de type **Byte** qui est parametre2. Un paramètre optionnel doit être énoncé **APRÈS** les paramètres obligatoires.

Encore plus loin...

Et oui on peut toujours aller plus loin ! Dans ce dernier point, on va se rendre compte qu'on peut envoyer à une fonction une variable de deux façon différentes. Soit on envoie la variable elle-même, soit une copie.

Pourquoi envoyer une copie de la variable ?

Lorsque vous envoyez une variable dans une fonction ou sous-routine, vous souhaitez faire des calculs avec ce nombre mais pas forcement modifier ce nombre, dans ce cas on envoie une copie de la variable à la fonction.

Pour envoyer une copie de cette variable, on va utiliser le mot-clé ByVal. De ce fait, la valeur réelle de la variable n'est pas modifiée. Si on utilise le mot-clé ByRef, on envoie la référence de la variable (l'adresse), la sous-routine peut alors accéder à la vraie valeur et la modifier. Dans ce cas, la variable est modifiée même dans la procédure qui appelle la sous-routine ou la fonction. Par défaut, les arguments sont passés par référence.

Nous allons faire un test :

```
' Passe la référence en argument.

Sub MaProcedure_1(ByRef x As Integer)

 x = x * 2

End Sub



' Passe la valeur en argument.

Sub MaProcedure_2(ByVal y As Integer)

 y = y * 2
```

```
End Sub



' ByRef est la valeur par défaut si non spécifiée.

Sub MaProcedure_3(z As Integer)

 z = z * 2

End Sub



Sub Test()



 ' On utilise une variable pour la faire changer

 Dim nombre_a_change As Integer

 nombre_a_change = 50



 ' On applique à notre variable la première sous-routine

 MaProcedure_1 nombre_a_change

 MsgBox nombre_a_change



 ' On applique à notre variable la deuxième sous-routine

 MaProcedure_2 nombre_a_change

 MsgBox nombre_a_change



 ' On applique à notre variable la troisième sous-routine

 MaProcedure_3 nombre_a_change

 MsgBox nombre_a_change
```

```
End Sub
```

Dans un premier temps, on applique à notre variable la première procédure qui multiplie par 2 la valeur. Étant donné que la variable est passée grâce à la référence, la variable est modifiée dans la sous-routine mais aussi dans la procédure qui appelait la sous-routine et donc on affiche le double de le valeur de départ, c'est à dire 100.

Dans un second temps, on applique la deuxième sous-routine. Dans celle-ci, on envoie une copie de la valeur (on envoie donc la valeur 100). Elle fait le même calcul mais avec une copie. De ce fait, la procédure qui appelait la sous-routine n'affiche pas le double mais 100. En effet, c'est la copie qui a été multipliée par 2 et non la valeur réelle de la variable.

Enfin, dans la troisième, aucun mot-clé n'est utilisé. Par défaut nous envoyons la référence et donc la vraie valeur à la sous-routine. Celle-ci effectue donc le même calcul (multiplication par 2) et modifie la variable de la procédure appelante. Le résultat affiché est donc 200.

C'est bon, c'est fini ! Euh, je parle de ce chapitre, pas du tutoriel, nous en sommes encore assez loin d'ailleurs  Les fonctions de ce chapitre ne sont pas très développées mais c'est en pratiquant que vous aller complexifier vos fonctions et sous-routines. La lecture des chapitres suivants vous aidera à faire tout ce que vous voulez facilement.