

# VBA : Programmer efficacement Microsoft Excel

## Notions importantes

### Les collections

On entend par collection, la réunion d'éléments connexes permettant d'accéder à ces éléments par leurs numéros d'index (ou parfois par leur nom), mais aussi d'ajouter ou de supprimer des membres.

Le modèle objet d'Excel contient ainsi de nombreuses collections hiérarchisées (on peut dire que l'application contient une collection de classeurs dont chaque élément contient une collection de feuilles etc...)

Une collection contient généralement au moins trois méthodes et une propriété de base.

Les méthodes Add et Remove (ou Delete) permettent l'ajout / suppression d'éléments, la méthode Item permet d'accéder à un élément par son index ou son nom, la propriété Count donne le nombre d'éléments de la collection. Attention toutefois, certaines collections sont de base 0, d'autres de base 1. Dans Excel, les collections sont de Base 1. Cela signifie que le numéro d'index 1 est le premier numéro d'index utilisable pour pointer sur un élément à travers la collection.

En général, la collection a le même nom que les objets qu'elle contient en y ajoutant un "s" à la fin. Par exemple la collection des objets classeurs Workbook s'appelle WorkBooks

Quelques remarques encore, les collections coûtent relativement cher en mémoire, donc avant d'utiliser un des membres fréquemment, il convient de l'affecter à une variable fortement typée.

Exemple:

```
Dim MaPlage as Range
Set MaPlage = ActiveSheet.Cells(1,1)
```

L'utilisation de "MaPlage" permet un meilleur accès à la cellule que l'utilisation d'ActiveSheet.Cells(1) tout au long du code.

L'appel à la méthode Item est implicite,

WorkBooks("NomClasseur").Activate et WorkBooks.Item("NomClasseur").Activate sont équivalents.

### L'énumération

Cette technique permet de parcourir une collection. Sa syntaxe est For Each....Next

Là encore, typez fortement votre variable d'énumération.

```
Dim MaCellule As Range
For Each MaCellule in ActiveSheet.Range("A1:E5")
....
Next
```

Dans l'exemple ci-dessus, "Dim MaCellule As Range" est mieux que "Dim MaCellule As Object" qui est mieux que "Dim MaCellule As Variant".

### L'Adressage

Excel fonctionne principalement par un système d'adresse (appelé aussi référence) pour localiser une cellule particulière. Le nom complet d'une adresse Excel valide est dans le cas d'une cellule :

*'[NomClasseur.xls]NomFeuille'!\$PositionCellule*. Excel utilise indifféremment deux types de références appelées A1 où les colonnes sont des lettres et les lignes des nombres, et L1C1 où lignes et colonnes sont des nombres.

L'exemple ci-dessous donne deux adresses Excel complètes dans les deux styles de références :

'[NomClasseur.xls]NomFeuille'!\$A\$35 ou '[NomClasseur.xls]NomFeuille'!L1C35

Ces références peuvent être relatives ou absolues. Une référence absolue donne la position d'une cellule dans la feuille par l'intersection de la ligne et de la colonne, une référence relative donne l'adresse de la cellule par décalage de l'adresse par rapport à une autre cellule. Une référence peut être relative dans une direction (par

exemple les colonnes) et absolue dans l'autre. Dans le tableau suivant, nous voyons l'écriture de l'adresse "B2" dans la cellule "A1" dans tous les modes

Références		A1	L1C1
Ligne	Colonne		
Absolue	Absolue	« =\$B\$2 »	L2C2
Absolue	Relative	« =B\$2 »	L2C(1)
Relative	Absolue	« =\$B2 »	L(1)C2
Relative	Relative	« =B2 »	L(1)C(2)

A l'identique en VBA on peut donner l'adresse d'un objet appartenant au modèle objet Excel par :  
Application.Workbooks("NomClasseur").Sheets("NomFeuille").Objet ...

Dans la pratique on ne met jamais Application dans l'adresse puisqu'il est clairement implicite (sauf dans le cas d'un classeur partagé dans un programme utilisant plusieurs instances d'Excel).

### ThisWorkbook

L'objet ThisWorkbook représente le classeur qui contient la macro qui est en train de s'exécuter Ceci sousentend, qu'il n'est jamais nécessaire de mettre dans une variable le classeur contenant la macro pour y faire référence.

### Objet actif

La notion de l'objet actif permet d'accéder à des raccourcis dans l'adressage VBA. Cette notion, très employée par l'enregistreur de macro est certes performante, mais non dénuée de risque. A chaque instant, de l'application, on peut nommer le classeur actif (visible) ActiveWorkbook, la feuille visible de ce classeur ActiveSheet et la cellule sélectionnée ActiveCell. (il y en a d'autres mais voilà les principaux)

Ceci permet bien des raccourcis d'écriture de code mais demande deux choses :

- Bien savoir quel est l'objet actif
- Manipuler l'activation

Or comme nous allons le voir au cours de cet article, manipuler l'activation est une méthode lourde et assez lente.

### Sélection

Cet objet représente ce qui est sélectionné. Cette simple définition devrait suffire pour se méfier de son utilisation. En effet, on l'utilise en général quand on ne sait pas quel est le type de l'objet que l'on cherche à atteindre. A mon avis son utilisation est à proscrire. Là encore, on le trouve très souvent dans les macros enregistrées, je vais donc vous donner une petite astuce pour connaître le type d'un objet. En général, tout objet inséré par le code, l'est par une méthode Add. On ajoute donc une variable de type variant où l'on affecte l'objet créé. La fonction TypeName permet alors de connaître le type de l'objet. Ceci peut se faire aussi avec l'espion express du débogueur.

### Paramètres nommés

Excel accepte une syntaxe particulière pour ses méthodes, celle des paramètres (ou arguments) nommés. En Visual Basic l'appel d'une procédure paramétrée se fait en plaçant les paramètres dans le même ordre que celui qui se trouve dans la déclaration de la procédure. Cette méthode reste vraie en VBA mais il y a aussi la possibilité de ne passer que quelques-uns de ces paramètres en les nommant. Regardons par exemple la méthode

Find d'Excel.

Sa déclaration est :

```
expression.Find(What, After, LookIn, LookAt, SearchOrder, SearchDirection, MatchCase, MatchByte)
```

Son utilisation normale serait

```
Set CelluleCible=ActiveSheet.Cells.Find(2,ActiveCell, xlValues, xlWhole, xlByColumns, xlNext, False)
```

Mais je pourrais très bien simplifier en écrivant

```
Set CelluleCible=ActiveSheet.Cells.Find(What:=2, LookIn:= xlValues)
```

Ceci permet une plus grande clarté du code ainsi qu'une simplification à la condition expresse de bien connaître la valeur par défaut des paramètres.

## Les évènements

La gestion des événements dans Excel se fait via du code soit dans un module d'objet WorkSheet, soit dans le module de l'objet Workbook. Pour atteindre le module de code du classeur, on ouvre la fenêtre VBA et dans l'explorateur de projet on double click sur "ThisWorkbook". Pour atteindre le module d'une feuille on peut soit passer par l'explorateur de projet, soit faire un click droit sur l'onglet de la feuille et choisir "Visualiser le code".

### Intercepter les évènements Excel en Visual Basic

Dans la feuille VB où l'on souhaite récupérer l'événement on déclare une variable globale

```
Private WithEvents MaFeuille As Excel.Worksheet
```

Après il suffit d'écrire une procédure d'évènements identique à la procédure Excel correspondante.

Par exemple, l'événement de feuille SelectionChange s'écrit dans Excel

```
Private Sub Worksheet_SelectionChange(ByVal Target As Excel.Range)
```

Et dans Visual Basic :

```
Private Sub MaFeuille _SelectionChange(ByVal Target As Excel.Range)
```

L'exemple suivant montre l'interception de l'événement BeforeClose d'un classeur dans Visual Basic afin d'empêcher la fermeture de celui-ci (et donc de l'application Excel) par l'utilisateur. Pour qu'Excel puisse se fermer dans cet exemple, il faut remplir la case "A1", ce qui est géré par l'événement

```
MonClasseur_SheetChange.
```

```
Private WithEvents MonClasseur As Excel.Workbook
```

```
Private MonExcel As Excel.Application, MaFeuille As Excel.Worksheet
```

```
Private Sub OuvreExcel()
```

```
Set MonExcel = New Excel.Application
```

```
MonExcel.ReferenceStyle = xlR1C1
```

```
Set MonClasseur =
```

```
MonExcel.Workbooks.Open("D:\User\jmarc\tutorial\excel\tutor1.xls")
```

```
Set MaFeuille = MonClasseur.Worksheets("pilotage")
```

```
MonExcel.Visible = True
```

```
End Sub
```

```
Private Sub MonClasseur_BeforeClose(Cancel As Boolean)
```

```
Cancel = True
```

```
End Sub
```

```
Private Sub MonClasseur_SheetChange(ByVal Sh As Object, ByVal Target As Excel.Range)
```

```
If Not MonExcel.Intersect(Target, MaFeuille.Cells(1, 1)) Is Nothing
```

```
Then
```

```
MonExcel.EnableEvents = False
MonClasseur.Close False
Set MaFeuille = Nothing
Set MonClasseur = Nothing
MonExcel.Quit
Set MonExcel = Nothing
End If
End Sub
```

## La base, l'objet Application

Cet objet représente l'application Excel. Il est l'objet parent de tous les autres, et contient des propriétés, méthodes et événements très intéressants, que nous allons regarder maintenant.

### Événements

On utilise rarement les événements au niveau de l'application, pour la raison simple qu'ils ne sont pas accessibles directement. Pour pouvoir les utiliser, il faut dans le projet créer un module de classe dans lequel on met le code suivant :

```
Public WithEvents App As Application
```

Comme la plupart des événements gérés se retrouvent dans les objets classeur et feuille, nous les utiliserons plutôt à ce niveau.

### Propriétés

Je vais donner maintenant quelques propriétés utiles de l'objet application

**AskToUpdateLinks** : Si sa valeur est `False` la mise à jour des liaisons se fait sans appel d'une boîte de dialogue, qui sinon apparaît automatiquement.

**Calculation** (`xlCalculationAutomatic`, `xlCalculationManual`, `xlCalculationSemiautomatic`): Définit le mode de recalcul d'Excel. Il s'agit d'une propriété très importante pour l'optimisation du temps d'exécution. En effet, pour les feuilles contenant des formules de calcul, le mode de calcul automatique peut être très coûteux en temps, en effet Excel recalcule intégralement une feuille de calcul à chaque modification ou suppression. C'est pourquoi, en général on bloque le mode de calcul au départ en faisant :

```
Application.Calculation = xlCalculateManual
```

Puis on exécute les calculs lorsqu'on en a besoin avec la méthode `Calculate`

**CutCopyMode** (`False`,`xlCopy`,`xlCut`) : Définit si on utilise un mode par défaut copier ou couper. En fait, on l'utilise souvent sous la forme `Application.CutCopyMode=False` pour vider le presse papier.

**DisplayAlerts** : Si sa valeur est `False`, les messages d'avertissements d'Excel ne seront pas affichés.

★ Il faut toujours remettre la valeur à `True` en fin d'exécution

**Interactive** : Désactive toutes les interactions entre Excel et le clavier/souris.

★ Il faut toujours remettre la valeur à `True` en fin d'exécution

**International** : Renvoie des informations relatives aux paramètres régionaux et internationaux en cours. Cette propriété est en lecture seule. On s'en sert généralement pour connaître les séparateurs.

**ReferenceStyle** (`xlA1` ou `xlR1C1`): Permet de basculer entre les modes `L1C1` et `A1`. Il est à noter que cela change l'affichage des feuilles mais que cela peut aussi avoir une influence sur le code. En effet certaines formules comme les règles de validations doivent être écrites dans le même système de référence que celui de l'application.

**ScreenUpdating** : Permet de désactiver la mise à jour d'écran. Pour des raisons de vitesse d'exécution il est conseillé de toujours la désactiver.

★ Il faut toujours remettre la valeur à `True` en fin d'exécution, sous peine de récupérer un écran figé.

**SheetsInNewWorkbook** : Permet de définir le nombre de feuilles contenues dans un nouveau classeur. Lorsque l'on sait exactement le nombre de feuilles nécessaires, on peut modifier cette propriété afin de ne pas faire des worksheets.add. Il convient de restaurer cette valeur en fin d'exécution.

## Méthodes

**Calculate** : Permet de forcer le calcul. La syntaxe Application.Calculate est peu utilisée. On l'utilise principalement sous la forme MaFeuille.Calculate. Sachez toutefois que l'on peut restreindre le calcul à une plage à des fins de performance. Exemple :  
Worksheets(1).Rows(2:3).Calculate ne fait les calculs que sur les lignes 2 et 3.

**Evaluate** : Permet de convertir une chaîne en sa valeur ou en l'objet auquel elle fait référence. Nous allons regarder quelques utilisations de cette méthode.

Interprétation de formule de calcul Imaginons que ma cellule A1 contient le texte (12\*3)+4, écrire  
Range("A2").Value=Application.Evaluate(Range("A1").Value)

Renverra 40 en A2. De même on pourra écrire :

Resultat= Application.Evaluate("(12\*3)+4")

La méthode permet aussi d'évaluer une formule respectant la syntaxe Excel (en anglais) ; on peut écrire  
Resultat= Application.Evaluate("Sum(A1:E5)")

Interprétation d'une adresse Si ma cellule A1 contient B1:B2 je peux écrire

Application.Evaluate(Range("A1").Value).Font.Bold=True

Il est à noter que le mot Application est facultatif et on trouve parfois la notation

[A1].Font.Bold=True

Qui est strictement équivalente !

**FindFile** Permet de lancer une recherche de fichier dans laquelle on spécifie les critères. La collection FoundFiles contient les fichiers trouvés correspondant.

(cf l'exemple plus loin)**GetOpenFileName** : Ouvre la boîte de dialogue "Ouvrir un fichier" mais n'ouvre pas le fichier. La méthode renvoie juste le nom complet du fichier sélectionné.

**GetSaveAsFilename** : De même que précédemment mais avec la boîte "Enregistrer Sous..."

**Goto** : Je cite cette méthode pour vous mettre en garde. L'enregistreur de macro l'utilise lors de l'appel d'une plage nommée, elle sous-tend un "Activate" et un "Select". Il faut donc faire très attention lors de son utilisation dans le code car elle peut facilement changer les objets actifs.

**Intersect** : Renvoie une plage qui est l'intersection de n plages. Bien que les plages appartiennent à des objets feuilles, la méthode Intersect appartient directement à l'objet Application.

**OnKey** : Permet l'exécution d'une macro lors de l'appui sur une touche ou sur une combinaison de touches.

Application.OnKey "^{A}", "MaProc" lance la procédure MaProc sur l'appui de CTRL-A

Pour restaurer la séquence de touche on écrit :

Application.OnKey "^{A}"

**OnTime** : Permet l'exécution d'une macro à une heure fixée ou après un délai précisé.

Application.OnTime Now + TimeValue("00:00:15"), "MaProc" attend 15 secondes avant d'exécuter la procédure MaProc

**Run** : Lance l'exécution de la procédure spécifiée. Si celle-ci attend des paramètres, ils ne peuvent pas être nommés.

Par exemple imaginons la fonction suivante :

```
Private Function MaSomme(Oper1 as Double, Oper2 as Double) As Double.
```

La syntaxe d'appel sera :

```
Resultat=Application.Run(MonClasseur!MaSomme,1.2,2.3)
```

**Union** : Renvoie l'union de n plages. Même remarque que pour Intersect.

**Wait** : Marque une pause dans l'exécution de la macro.

## Collections et objets particuliers

Nous allons regarder trois collections (objets) intéressantes accessibles depuis l'objet Application.

### FileSearch

L'objet FileSearch permet une recherche standard de fichiers. On peut spécifier ses critères de recherche en valorisant ses propriétés. L'appel de la méthode Execute lance la recherche. Le résultat se trouve alors dans la collection FoundFiles.

Par exemple pour rechercher l'ensemble des fichiers Res\*.xls dans le répertoire courant, et les ouvrir.

```
Public Sub OuvreFichier()  
Dim ChercheFichier As FileSearch, compteur As Long  
Set ChercheFichier = Application.FileSearch  
With ChercheFichier  
.LookIn = CurDir  
.FileName = "Res*"  
.FileType = msoFileTypeExcelWorkbooks  
.SearchSubFolders = True  
.Execute  
For compteur = 1 To .FoundFiles.Count  
Workbooks.Open .FoundFiles(compteur)  
Next  
End With  
End Sub
```

### Dialogs

La collection Dialogs contient l'ensemble des boîtes de dialogues utilisées par Excel. Comme il y en a beaucoup (646 dans Excel 97) je ne vais pas les énumérer, vous trouverez cette liste soit dans l'explorateur d'objets, soit dans l'aide à "Listes d'arguments de boîte de dialogue intégrée", soit dans le fichier ListeVBA.xls que vous retrouverez dans votre dossier Office (VBAlist.xls depuis Excel 2000). Ces boîtes agissent le plus souvent sur la sélection en cours donc nous sommes dans un des rares cas où l'emploi de la méthode Select sera obligatoire.

Voici par exemple l'appel de la boîte de dialogue "alignement" pour permettre à l'utilisateur de modifier l'alignement sur la plage B1:B10

```
Public Sub UtilDial()  
Range(Cells(1, 2), Cells(10, 2)).Select  
Application.Dialogs(xlDialogAlignment).Show  
End Sub
```

On pourrait de la même façon donner des valeurs par défaut à cette boîte en lui passant des arguments

Par exemple :

```
Application.Dialogs(xlDialogAlignment).Show 3, False, 2
```

Ouvre la même boîte mais en définissant les alignements verticaux et horizontaux sur "centré".

### WorksheetFunction

Cet objet contient les fonctions de feuille de calcul intégrées dans Excel. Attention ces fonctions sont en anglais. Il faut faire très attention aux types des paramètres passés, en effet certaines fonctions acceptent indifféremment des plages ou des nombres alors que d'autres n'acceptent que des plages. L'exemple suivant montre le calcul d'une moyenne mêlant chiffres et plages

```
Dim resultat As Double  
resultat = Application.WorksheetFunction.Average(Range(Cells(1, 2),  
Cells(10, 2)), 100, 200)
```

Là encore je ne donnerai pas la liste complète des fonctions que vous pourrez trouver dans l'aide, dans le

fichier listeVBA.xls ou avec l'explorateur d'objets.

## Résumé

Plus loin dans cet article nous trouverons des exemples d'utilisation des propriétés/méthodes de l'objet Application, mais il faut bien garder à l'esprit qu'une procédure Excel devrait toujours contrôler le mode de calcul et désactiver la mise à jour de l'écran.

## L'objet Workbook (classeur)

Le classeur est en général l'objet central de la programmation Excel. Lorsque l'application va utiliser un seul classeur, on peut utiliser ActiveWorkbook, mais dans le cas d'une application multi-classeurs, il convient de les mettre dans des variables afin d'en simplifier la manipulation. Pour cela, on fait :

### En VBA

```
Dim MonClasseur as Workbook  
Set MonClasseur=ActiveWorkbook
```

### En VB

```
Dim MonClasseur as Excel.WorkBook  
Set MonClasseur=ActiveWorkbook
```

On peut bien sûr faire l'affectation dans le même temps que l'ouverture ou l'ajout, par exemple

```
Set MonClasseur=WorkBooks.Open FileName:="C:\User\Classeur1.xls"
```

N.B. (*en VBA uniquement*) : Si on utilise deux classeurs dont celui qui contient la macro, il n'est pas utile de mettre celui-ci dans une variable puisqu'on le retrouve sous le nom "ThisWorkbook".

## La collection WorkBooks

Cette collection contient l'ensemble des classeurs **ouverts**. L'ordre des classeurs dans la collection (index) est l'ordre d'ouverture. Elle possède quatre méthodes que nous allons étudier

### Add

Permet d'ajouter un nouveau classeur à la collection et non pas d'ouvrir un classeur existant. Le classeur créé devient le classeur actif.

Elle suit la syntaxe *Workbooks.Add(Template)*

Si Template est un fichier Excel existant, le classeur est créé en suivant le modèle défini. Template peut aussi être une constante (xlWBATChart, xlWBATExcel4IntlMacroSheet, xlWBATExcel4MacroSheet, xlWBATWorksheet) et dans ce cas, le classeur ne contiendra qu'une feuille du type donné. Enfin si Template est omis un nouveau classeur standard est créé.

### Close

Cette méthode ferme tous les classeurs de la collection. Donc attention de ne pas confondre WorkBooks.Close et WorkBooks(1).Close

### Open

Ouvre un classeur Excel. Sa syntaxe est :

```
WorkBooks.Open(FileName, UpdateLinks, ReadOnly, Format, Password, WriteResPassword,  
IgnoreReadOnlyRecommended, Origin, Delimiter, Editable, Notify, Converter, AddToMRU)
```

Seul FileName est obligatoire.

### OpenText

Permet d'ouvrir un fichier texte délimité comme un classeur. Sa syntaxe est

```
WorkBooks.OpenText(Filename, Origin, StartRow, DataType, TextQualifier, ConsecutiveDelimiter, Tab,  
Semicolon, Comma, Space, Other, OtherChar, FieldInfo)
```

## Quelques méthodes à connaître

**Close**

Ferme le classeur. La syntaxe est :

*MonClasseur.Close(SaveChanges, FileName, RouteWorkbook)*

Si SaveChanges est omis, une boîte de demande d'enregistrement apparaît. Donner une valeur FileName différentes de celle du classeur revient à faire un SaveAs.

**PrintOut**

Lance l'impression du classeur. Sa syntaxe est :

*MonClasseur.PrintOut(from, To, Copies, Preview, ActivePrinter, PrintToFile, Collate)*

A noter que From et To sont des pages d'imprimante et non les feuilles du classeur, et que ActivePrinter permet de choisir l'imprimante

**Protect / UnProtect**

Active ou désactive la protection du classeur. La syntaxe est :

*MonClasseur.Protect(Password, Structure, Windows)* pour activer la protection

*MonClasseur.UnProtect(Password)* pour la retirer.

A ce propos, la protection du classeur et des feuilles est une chose indépendante. La protection du classeur sert à bloquer la structure de celui-ci, celle de la feuille à protéger les objets qu'elle contient. On peut parfaitement protéger une feuille sans protéger le classeur et inversement.

**Save / SaveAs**

Sauvegarde le classeur. La syntaxe de SaveAs est :

*MonClasseur.SaveAs(FileName, FileFormat, Password, WriteResPassword, ReadOnlyRecommended, CreateBackup, AddToMru, TextCodePage, TextVisualLayout)*

**Evènements du classeur**

L'objet classeur possède beaucoup d'évènements, je ne vais donc détailler que ceux les plus souvent utilisés.

**BeforeClose, BeforePrint, BeforeSave**

*Private Sub Workbook\_Before----(Cancel As Boolean)*

Se produit avant l'évènement. Dans le cas de BeforeClose se produit avant la demande d'enregistrement des modifications. Mettre Cancel = True dans la procédure empêche l'évènement d'avoir lieu.

**Open**

Se produit à l'ouverture du classeur. Permet donc de faire des macros à exécution automatique

**SheetBeforeDoubleClick, SheetBeforeRightClick**

*Private Sub MonClasseur\_SheetBefore----Click(ByVal Sh As Object, ByVal Target As Range, ByVal Cancel As Boolean)*

Ces évènements sont des évènements de feuille. On utilise l'évènement au niveau classeur lorsqu'on veut centraliser l'évènement pour plusieurs feuilles du classeur. S'il existe aussi une procédure pour l'évènement au niveau feuille, elle s'exécutera avant la procédure du classeur.

Cet évènement ne se produit pas sur les feuilles graphiques.

L'objet "Sh" est un objet WorkSheet qui représente la feuille sur laquelle l'évènement se produit. Nous regarderons l'argument Target dans "les techniques de programmation".

**SheetCalculate, SheetChange, SheetSelectionChange**

Les mêmes remarques sont valables. Nous étudierons en détail ces évènements dans le chapitre sur les feuilles.

**Les Feuilles (Sheets)**

Un classeur Excel est composé d'une ou plusieurs feuilles. Ces feuilles peuvent être de cinq types :

Feuille de calcul, Feuille graphique, Macro Excel4, Macro Excel4 International, feuilles de boîtes de dialogue Excel 5.

La collection Sheets contient toutes les feuilles du classeur quel que soit leur type. En fait, on utilise très



peu cette collection sauf pour connaître le nombre total de feuille avec la propriété Count ou parfois pour ajouter une feuille de type macro Excel4.

## Feuille de calcul (WorkSheet)

L'objet WorkSheet doit suivre les mêmes règles de programmation que l'objet classeur. On utilise ActiveSheet que dans le cas d'une application ne mettant en jeu qu'une seule feuille. Sinon on référence la feuille par :

### En VBA

```
Dim MaFeuille as Worksheet
Set MaFeuille=ActiveSheet
```

### En VB

```
Dim MaFeuille as Excel.WorkSheet
Set MaFeuille=ActiveSheet
```

Observons qu'il est rarement nécessaire de référencer à la fois la feuille et le classeur.

## La collection Worksheets

Cette collection contient l'ensemble des feuilles de calcul du classeur. Le numéro d'ordre d'un objet Worksheet dans cette collection est son ordre dans les onglets du classeur.

### Méthodes utiles

#### *Add Worksheet.Add(Before, After, Count, Type)*

*Before* et/ou *After* permettent de préciser la position où la feuille doit être ajoutée. L'argument *Count* détermine le nombre de feuilles à ajouter. *Type* revient à faire un Add de la collection Sheets.

Comme je l'ai signalé au préalable, la position de l'objet Worksheet dans l'ensemble des Worksheets du classeur va aussi être son index dans la collection Worksheets. C'est pourquoi accéder à une feuille par Worksheets(index) peut être une source d'erreur.

#### **Copy** Worksheets(index).Copy(Before, After)

Duplique une feuille dans le classeur ou crée une copie dans un autre classeur.

Attention lors de la copie d'une feuille, la collection "Names" des plages nommées appartient à l'objet Workbook et à l'objet Worksheet. Lors de la duplication de la feuille, Excel va créer deux noms presque identiques, ce qui peut engendrer des bugs.

#### **Delete** Worksheets(index).Delete

Supprime la feuille spécifiée par index. Un classeur doit toujours contenir au moins **une** feuille.

#### **FillAcrossSheets** Worksheets(collection). FillAcrossSheets (Range, Type)

Permet de recopier une plage sur plusieurs feuilles en même temps. *Range* détermine la plage à copier, *Type* définit le mode de copie (xlFillWithAll, xlFillWithContents ou xlFillWithFormulas)

Il faut passer une collection ou un tableau d'objets Worksheet contenant les feuilles concernées par la copie à la méthode. Cette collection doit toujours contenir la feuille contenant la plage source.

Le code suivant recopie la plage A1:A10 sur toutes les feuilles de calcul du classeur

```
Dim MaFeuille As Worksheet
Set MaFeuille = ActiveWorkbook.Worksheets(1)
Worksheets.FillAcrossSheets MaFeuille.Range(Cells(1, 1), Cells(10, 1)),
xlFillWithAll
```

Le code suivant recopie la même plage dans la feuille "Feuil3"

```
Dim MaFeuille As Worksheet, TabFeuille As Variant
Set MaFeuille = ActiveWorkbook.Worksheets(1)
TabFeuille = Array("Feuil1", "Feuil3")
Worksheets(TabFeuille).FillAcrossSheets MaFeuille.Range(Cells(1, 1),
Cells(10, 1)), xlFillWithAll
```

#### **Move** Worksheets(index).Move(Before, After)

Similaire à copy mais déplace la feuille. Dans ce cas il n'y a pas de problème avec les noms.

## Evènements de l'objet WorkSheet

### BeforeDoubleClick, BeforeRightClick

*Private Sub Worksheet\_BeforeDoubleClick(ByVal Target As Range, Cancel As Boolean)*

Se produit avant que l'effet du double click ou du click droit se produise. Mettre Cancel à True annule l'évènement normalement attendu. Target renvoie un objet Range qui représente la cellule ayant reçu l'évènement.

### Calculate

Se produit **après** le recalcul de la feuille.

### Change

*Private Sub Worksheet\_Change(ByVal Target As Range)*

Se produit lorsque le contenu d'une cellule ou d'une plage change, du fait de l'utilisateur ou d'un lien externe, ou du programme, mais pas par le recalcul.

Target renvoie la plage ou la cellule modifiée. Pour la gestion de ces plages voir plus loin au chapitre "techniques de programmation".

### SelectionChange

*Private Sub Worksheet\_SelectionChange(ByVal Target As Excel.Range)*

Se produit lorsque la sélection a changé. Target renvoie la nouvelle plage sélectionnée.

Attention à la programmation de cet évènement. La présence de méthode Select dans la procédure peut déclencher un évènement en cascade.

## Quelques propriétés

### Names

Renvoie la collection des noms spécifique à la feuille de calcul. Attention les plages nommées appartiennent à l'objet Workbook. La collection Names renvoyée par cette propriété ne contient donc pas les plages nommées contenues par la feuille sauf si l'on a défini ce nom comme spécifique.

N.B : Pour votre culture générale, lors de la définition d'un nom, si on écrit NomFeuille!Nom on crée une plage nommée spécifique.

### PageSetup

Renvoie un objet PageSetup qui contient toutes les informations de mise en page de la feuille.

### Shapes

Renvoie la collection Shapes de toutes les formes présentes sur la feuille. Cette collection peut contenir beaucoup d'objets de types différents. Je n'aborderai pas dans cet article la programmation avec Shapes.

### Visible

Affiche ou masque la feuille. La propriété peut prendre la valeur xlVeryHidden qui masque la feuille de telle façon qu'elle ne puisse être rendue visible que par le code ou par l'explorateur de projet.

## Plage et cellule (Range)

Nous allons aborder ici la clé de la programmation. Une plage de cellules (objet Range) représente n'importe quel groupe de cellules d'une feuille. La plage peut être constituée d'une cellule, d'une plage continue ou de plusieurs plages. L'objet Range, lorsqu'il représente des cellules, appartient toujours à un objet Worksheet. L'objet Range ne possède pas d'évènement spécifique, néanmoins on peut lui faire utiliser les évènements de feuille ou de classeur.

Excel fonctionne avec un système d'adresse. Il existe en fait deux styles de référence :

R1C1 Les colonnes et les lignes sont représentées par des nombres

A1 Les colonnes sont des lettres et les lignes des nombres.

Le changement de style au cours d'un programme ne pose pas de problème vis-à-vis des formules déjà existantes dans une feuille. Sachez toutefois que le code doit être rédigé dans le même style de référence que l'application, sous peine de voir parfois se déclencher des erreurs.

En mode A1 on utilise un range comme suit :

MaFeuille.Range("A1") : une cellule

MaFeuille.Range("A1:E10") ou Range("A1", "E10") : une plage continue

En mode RIC1

MaFeuille.Cells(1,1) : une cellule

MaFeuille.Range(Cells(1,1),Cells(10,5)) : une plage continue

Quel que soit le style, les plages discontinues s'obtiennent à l'aide de la méthode Union que nous verrons plus en détail dans la deuxième partie.

En général j'utilise la syntaxe RIC1. La notation est Cells(Ligne, Colonne)

Je vais faire maintenant un parcours de ses propriétés et méthodes.

## **Propriétés ne renvoyant pas un objet - collection**

### **Address**

*MonRange.Address(RowAbsolute, ColumnAbsolute, ReferenceStyle, External, RelativeTo)*

*RowAbsolute* et *ColumnAbsolute* détermine si la référence renvoyée est relative ou absolue, *ReferenceStyle* détermine le style de référence, *External* ajoute le nom du classeur et le nom de la feuille.

Comme nous l'avons vu, peu importe le style utilisé puisque la propriété Address renvoie la référence dans le style désiré.

### **Column, row**

Renvoie le numéro de la première colonne / ligne de la plage

### **ColumnWidth, RowHeight**

Renvoie ou définit la largeur / hauteur des colonnes / lignes de la plage.

### **Formula / FormulaRIC1**

Définit ou renvoie la formule de la plage, cette formule étant en anglais.

### **FormulaLocal**

Identique à Formula mais dans la langue de l'utilisateur.

### **Hidden**

Masque une ou plusieurs ligne(s)/colonne(s). Attention l'ensemble de la ligne ou de la colonne doit être sélectionnée (voir plus loin à "plage particulière").

### **HorizontalAlignment**

Définit l'alignement dans la plage. Peut permettre un centrage multi-colonnes si la plage contient plusieurs colonnes.

### **Locked**

Renvoie ou définit si les cellules sont verrouillées. Le verrouillage n'a aucun effet tant que la feuille n'est pas protégée.

### **MergeCells**

Renvoie Vrai si la cellule fait partie d'une plage fusionnée. A ce propos, évitez si possible de fusionner les cellules. C'est très souvent une source de problèmes.

### **Name**

Renvoie ou définit le nom de la plage. Comme nous le verrons dans les techniques de programmation, les plages nommées sont très utiles.

### **NumberFormat**

Renvoie ou définit le format des cellules de la plage. Lors de la lecture, renvoie NULL si tous les formats ne sont pas identiques.

### **ShrinkToFit**

Force le contenu à s'adapter aux dimensions de la cellule.

**WrapText**

Force le retour à la ligne si le contenu dépasse la largeur de la cellule.

**Propriétés renvoyant un objet****Areas**

Renvoie la collection Areas de l'objet Range. Si la plage est une plage continue, la collection Areas ne contient qu'un élément qui est l'objet Range. Si cette plage contient plusieurs plages discontinues, il y a autant d'éléments que de plages continues dans l'objet Range.

Par Exemple

```
Dim MaPlage As Range, NbPlage As Integer, AdrTest As String
Set MaPlage = Union(Range(Cells(1, 1), Cells(5, 1)), Range(Cells(1, 3),
Cells(5, 3)), Range(Cells(1, 5), Cells(5, 5)))
NbPlage = MaPlage.Areas.Count
AdrTest = MaPlage.Areas(2).Address(True, True, xlA1)
Dans ce cas NbPlage renvoie 3 et AdrTest = "$C$1:$C$5"
```

**Borders**

Renvoie une collection des objets Border d'une cellule ou d'une plage.

On peut manipuler tous ces objets en même temps avec un appel à Borders ou en spécifier un avec sa propriété Item.

L'exemple ci-dessous crée un encadrement léger intérieur, avec un contour plus épais :

```
With Range(Cells(1, 1), Cells(5, 1)).Borders
.LineStyle = xlContinuous
.Item(xlEdgeBottom).Weight = xlMedium
.Item(xlEdgeLeft).Weight = xlMedium
.Item(xlEdgeTop).Weight = xlMedium
.Item(xlEdgeRight).Weight = xlMedium
End With
```

**Cells**

Cette propriété renvoie un objet Range (une cellule) avec des coordonnées relatives à la première cellule de l'objet Range. Il faut faire très attention à cette erreur relativement fréquente.

MaFeuille.Cells(3,3) représente la cellule "C3", par contre si mon objet range est B1:B3 alors

MaRange.Cells(3,3) représente la cellule "D3", c'est à dire celle qui est en position (3,3) par rapport à la cellule "B1".

**Characters**

Renvoie un objet Characters sur le contenu d'une cellule ou sur les objets Shapes (Ne renvoie rien si la plage contient plusieurs cellules).

Un objet Characters renvoie tout ou partie (comme un Mid) du contenu d'une cellule, à condition que cela soit du texte.

La syntaxe est MaCellule.Characters(Start,Length).

Ainsi le code suivant passe les caractères 2 et 3 de la chaîne contenue dans la cellule en police "Symbol"

```
Cells(1, 7).Characters(2, 2).Font.Name = "Symbol"
```

Il est à noter que la modification de la police est la seule utilisation que j'ai rencontré de l'objet Characters dans une cellule.

**Columns / Rows**

Renvoie la collection de toutes les colonnes / lignes contenues dans la plage. Cela peut permettre certains raccourcis de programmation intéressant. Par exemple :

```
MaPlage.Columns(2).Value=""
```

Efface le contenu de toutes les cellules de la colonne 2 **dans la plage**.

## CurrentArray

Cette propriété est un peu particulière. Si la plage fait partie d'une formule matricielle, CurrentArray renvoie une plage contenant toutes les cellules de cette formule matricielle.

## CurrentRegion

Renvoie la plage en cours dans laquelle est l'objet Range. On entend par plage en cours, l'ensemble des cellules limitées par une combinaison de lignes et de colonnes vides.

Habituellement, on utilise cette propriété avec une cellule. Regardons l'exemple suivant :

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	3	4	3				0	2	6
2	0	1	3	4	3				0	2	6
3	0	1	3	4	3				0	2	6
4	0	1	3	4	3				0	2	6
5	0	1	3	4	3				0	2	6
6											
7											
8											

Les commandes suivantes renvoient respectivement

```
ActiveSheet.Cells(2, 2).CurrentRegion.Address(True, True, xlR1C1)
```

R1C1:R5C5

```
ActiveSheet.Cells(5, 10).CurrentRegion.Address(True, True, xlR1C1)
```

R1C9:R5C11

```
ActiveSheet.Cells(4, 7).CurrentRegion.Address(True, True, xlR1C1)
```

R4C7

```
ActiveSheet.Cells(4, 6).CurrentRegion.Address(True, True, xlR1C1)
```

R1C1:R5C6

Les deux premiers cas sont faciles à comprendre.

Pour le troisième comme la cellule (4,7 ou "G4") est entourée de cellules vides, CurrentRegion renvoie la cellule.

Le dernier cas est le plus intéressant. La cellule est vide mais contiguë à une plage non vide. Dans ce cas CurrentRegion renvoie un objet Range rectangulaire contenant la cellule cible et la plage contiguë. Cette contiguïté peut être diagonale.

## Dependents / DirectDependents/ Precedents / DirectPrecedents

Renvoie un objet Range contenant les cellules dépendantes ou antécédentes. Une cellule antécédente est une cellule qui contient dans sa formule de calcul la référence à la cellule cible. Une cellule dépendante est une cellule dont la référence est contenue dans la formule de calcul de la cellule cible. L'objet Range renvoyé peut être une plage multiple. Cette propriété peut être extrêmement intéressante pour déterminer quelle plage doit être recalculée. Nous verrons des utilisations dans la deuxième partie, mais je vais vous donner un exemple.

	4	5	6	7
7				
8		12		
9	15		=L9C5+L8C5	
10	20			=L9C6+L11C6
11	21		=L11C5+L10C5	
12				

regardons le code suivant :

```
Dim Recup As Range, raTest As Range
Set Recup = ActiveSheet.Cells(10, 7).Precedents
For Each raTest In Recup.Areas
```

```
Debug.Print raTest.Address(True, True, xlR1C1)
Next
```

Le résultat dans la fenêtre d'exécution sera

R9C6

R11C6

R8C5:R11C5

Si j'avais utilisé DirectPrecedents, seuls les deux premiers résultats aurait été renvoyés.

S'il n'y a pas de cellules correspondantes, la propriété déclenche une erreur récupérable.

## End

Cette propriété renvoie un objet Range décalé dans le sens spécifié à l'intérieur de la région. Toujours dans mon exemple précédent, `ActiveSheet.Cells(2, 2).End(xlDown).Address(True, True, xlR1C1)` renvoie R5C2 qui est la dernière cellule remplie vers le bas de la colonne 2.

Si la cellule sélectionnée est vide, est que la colonne est vide, la cellule renvoyée est la dernière cellule de la feuille, c'est à dire :

`ActiveSheet.Cells(2, 6).End(xlDown).Address(True, True, xlR1C1)` renvoie R65536C6.

Il y a un piège avec cette propriété. Supposons que la colonne 12 contienne une valeur dans la ligne 1 et que toutes les autres cellules soient vides, alors

`ActiveSheet.Cells(1, 12).End(xlDown).Address(True, True, xlR1C1)` renverra R65536C12

## Font

Renvoie ou définit un objet Font pour la plage. Cet objet permet de modifier la police, ainsi que ses propriétés. Attention toutefois, la police renvoyée lors de la lecture d'une plage affectera NULL aux propriétés qui ne sont pas les mêmes sur toute la plage. Supposons que la cellule (2,2) soit en gras. Si je fais :

```
Dim MaPlage As Range, LaPolice as Font
```

```
Set MaPlage = Range(Cells(1, 2), Cells(3, 2))
```

```
Set LaPolice = MaPlage.Font
```

Dans ce cas `LaPolice.Bold` sera NULL puisque toutes les cellules ne sont pas en gras.

## Interior

Renvoie ou définit un objet Interior pour la plage. Cet objet représente le motif et la couleur de l'intérieur de la cellule. Comme pour l'objet Font, il renvoie NULL pour les propriétés qui ne sont pas toutes identiques dans la plage.

## Offset

Renvoie un objet range décalé par rapport à l'objet sur lequel on fait l'offset. Sa syntaxe est :

`MaPlage.Offset(Offset de ligne, Offset de Colonne)`.

Par Exemple :

```
Range(Cells(1, 1), Cells(5, 1)).Offset(2, 2).Address(True, True, xlR1C1) renvoie R3C3:R7C3
```

On peut omettre un des paramètres s'il n'y a pas de décalage, mais pour la lisibilité du code, il convient de mettre 0.

## Resize

Renvoie un objet range redimensionné par rapport à la plage d'origine.

La syntaxe est `MaPlage.Resize(NbLigne, NbColonne)`

Comme on redimensionne la plage, c'est la cellule en haut à gauche de la plage d'origine qui sert de base.

Pour ne pas redimensionner dans une des dimensions, on omet le paramètre. Par exemple

```
Range(Cells(1, 1), Cells(5, 1)).Resize(2, 2).Address(True, True, xlR1C1) renvoie R1C1:R2C2
```

```
Range(Cells(1, 1), Cells(5, 1)).Resize(, 2).Address(True, True, xlR1C1) renvoie R1C1:R5C2
```

La combinaison d'Offset et de Resize permet de définir n'importe quelle plage.

## Quelques méthodes

Nous allons maintenant regarder quelques-unes des méthodes les plus utiles de l'objet Range.

**AutoFill**

Syntaxe `MaPlage.AutoFill(Destination, Type)`

Destination est un objet Range qui contient forcément MaPlage

Type peut être `xlFillDefault`, `xlFillSeries`, `xlFillCopy`, `xlFillFormats`, `xlFillValues`, etc...

La méthode exécute une recopie **incrémentée** dans les cellules de la plage destination.

**AutoFit**

Ajuste la largeur de colonne / hauteur de ligne en fonction du contenu.

S'applique sur des colonnes ou des lignes mais pas directement sur la plage.

`Range(Cells(1, 2), Cells(3, 2)).AutoFit` renvoie une erreur, il faut écrire

`Range(Cells(1, 2), Cells(3, 2)).Columns.AutoFit`

**BorderAround**

Syntaxe `MaPlage.BorderAround(LineStyle, Weight, ColorIndex, Color)`

Permet de réaliser directement un contour extérieur d'une plage. Ainsi l'exemple que j'ai donné pour la propriété `Borders` s'écrirait plus simplement :

```
With Range(Cells(1, 1), Cells(5, 1))
.Borders.LineStyle = xlContinuous
.BorderAround Weight:=xlMedium
End With
```

**Calculate**

Bien que cette méthode appartienne à l'objet application, on l'utilise lorsqu'on veut minimiser le temps de calcul sur un objet range restreint. En général elle s'utilise en coordination avec un événement `Change` et une plage `Dependents`.

Nous pourrions imaginer la fonction suivante qui ne calculerait en permanence que les cellules nécessaires.

```
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
On Error GoTo EnleveErreur
Target.Dependents.Calculate
Exit Sub
EnleveErreur:
Err.Clear
End Sub
```

Ce code n'est qu'un exemple puisque c'est ce que fait Excel. Celui ci ne servirait que si on utilise des fonctions "volatiles", ou si on veut restreindre la plage de calcul.

**Clear / ClearContents / ClearFormats**

Efface tout ou la partie spécifiée de la plage.

**ColumnDifferences / RowDifferences**

Syntaxe : `MaPlage.ColumnDifferences(Comparison)`

Renvoie un objet Range contenant toutes les cellules de la plage dont le contenu est différent de celui spécifié dans l'argument `Comparison`. Quelques limitations toutefois, l'argument `comparison` doit être un objet Range contenant **une seule** cellule ; elle doit faire partie de la plage. L'objet Range renvoyé peut être discontinu, alors que la plage d'appel doit être continue. La comparaison ne se fait pas sur toute la feuille mais sur la plage `UsedRange` (voir plus loin)

Ces méthodes sont très puissantes dans un certains nombres de cas. Supposons que j'ai un tableau contenant des cellules vides dans ma feuille, le code suivant va me renvoyer un objet Range contenant toutes les cellules non vides de la feuille.

```
Dim MaPlage As Range, raEnum As Range
Set MaPlage =
```

```
ActiveSheet.Columns.ColumnDifferences(ActiveSheet.Cells.SpecialCells(xlCellTypeLastCell).Offset(1, 1))
For Each raEnum In MaPlage.Areas
```

```
Next
```

## Copy

Cette méthode utilise ou non l'argument Destination. Lorsque l'argument est omis, la méthode copie la plage dans le presse-papiers, sinon le collage à lieu dans le même temps. Sauf cas très particulier il faut toujours préciser l'argument Destination. Cet argument doit être soit un objet Range de la même dimension que la plage source, soit une cellule unique qui deviendra le coin supérieur gauche de la plage collée. Pour donner un exemple, le code généré par l'enregistrement automatique serait :

```
Range("A1:E5").Select
Selection.Copy
Worksheets("Feuil2").Select
Range("C3").Select
ActiveSheet.Paste
```

Ce qui revient à écrire :

```
Range("A1:E5").Copy Destination:=Worksheets("Feuil2").Range("C3")
```

Dans certaines versions d'Excel, une erreur se produit si une des plages contient des cellules fusionnées.

En Excel97 il est impératif que la zone copiée soit située sur la feuille active.

## Insert / Delete

Permet d'insérer ou de supprimer une plage de cellules. Utilise l'argument Shift pour définir le sens de déplacement des cellules. Pour insérer une ligne ou une colonne entière utiliser EntireRow / EntireColumn (voir plus loin)

## FillDown, FillLeft, FillRight, FillUp

Recopie la plage avec le contenu de la cellule source. La cellule source doit être à l'extrémité inverse du sens de la recopie.

Ainsi Range(Cells(1,1),Cells(10,1)).FillDown recopie la cellule A1 dans la plage

Range(Cells(1,1),Cells(10,1)).FillUp recopie la cellule A10 dans la plage

## Find & FindNext

La syntaxe de cette méthode est MaPlageFind(What, After, LookIn, LookAt, SearchOrder, SearchDirection, MatchCase, MatchByte)

Je vais en profiter pour tordre le cou à un canard. La méthode Find ne déclenche pas une erreur lorsqu'elle ne trouve pas de cellules correspondantes, comme je l'ai souvent lu dans des forums. Lors d'un enregistrement automatique on obtient :

```
Selection.Find(What:="27", After:=ActiveCell, LookIn:=xlFormulas, LookAt _
:=xlPart, SearchOrder:=xlByRows, SearchDirection:=xlNext, MatchCase:= _
False).Activate
```

Ainsi qu'un message d'alerte qui signale qu'il n'y a pas de cellules correspondantes. Lorsqu'on prend ce bout de code et qu'on cherche à le bricoler pour faire une recherche dans son programme, on constate qu'il se produit une erreur s'il n'y pas de cellule contenant la recherche. Or ce qui est en cause n'est pas la méthode Find, mais l'appel de la méthode Activate. En effet, Find renvoie un objet Range qui est la première cellule contenant le critère ou Nothing s'il n'y a pas.

Et l'appel de Nothing.Activate fait planter l'exécution.

Ce qui en une méthode me permet de montrer qu'il faut se méfier de l'enregistrement automatique, de la méthode Activate et des rigolos qui font des copier/coller de bout de code.

La méthode Find se programme comme dans l'exemple suivant.

```
Dim MaFeuille As Worksheet, Reponse As Range, PremAdresse As String
Dim MonCritere As String
Set MaFeuille = ActiveSheet
MonCritere = "27"
```



```

Set Reponse = MaFeuille.Cells.Find(MonCritere, MaFeuille.Cells(1, 1),
xlValue, xlWhole)
If Not Reponse Is Nothing Then
PremAdresse = Reponse.Address
Do
Set Reponse = Selection.FindNext(After:=Reponse)
Loop While Not Reponse Is Nothing And Reponse.Address <>
PremAdresse
End If

```

## PasteSpecial

Bien que l'on puisse souvent se passer de cette méthode, il y a des cas où elle est très utile. Sa syntaxe est :  
 MaPlage.PasteSpecial(Paste, Operation, SkipBlanks, Transpose)

## Replace

Fonctionne sur le même schéma que Find, à la différence que cette méthode ne renvoie rien.

## Sort

Tri la plage spécifiée. Sa syntaxe est :

```
MaPlage.Sort(Key1, Order1, Key2, Type, Order2, Key3, Order3, Header, OrderCustom, MatchCase,
Orientation)
```

Type n'est utilisé que pour les objets PivotTable.

Les Arguments Key sont des variants, contenant la clé de tri, les arguments Order sont le sens du tri.

Header définit s'il y a une ligne / colonne d'entête, MatchCase si le tri est sensible à la casse, Orientation donne le sens du tri.

Par exemple

```

Range("I1:K5").Sort Key1:=Range("K1"), Order1:=xlAscending,
Key2:=Range("J1") _
, Order2:=xlDescending, Header:=xlGuess, OrderCustom:=1,
MatchCase:= _
False, Orientation:=xlTopToBottom

```

## SpecialCells

```
MaPlage.SpecialCells(Type, Value)
```

Renvoie un objet Range contenant les cellules correspondant aux critères donnés en argument. Les choix possibles sont :

XlCellTypeNotes Les cellules contenant des annotations.

xlCellTypeConstants Les cellules contenant des constantes.

xlCellTypeFormulas Les cellules contenant des formules.

XlCellTypeBlanks Les cellules vides.

XlCellTypeLastCell La dernière cellule de la plage utilisée.

XlCellTypeVisible Toutes les cellules visibles.

Les Types xlCellTypeConstants et xlCellTypeFormulas acceptent un argument Value qui peut être xlErrors, xlLogical, xlNumbers, xlTextValues, xlAllFormatConditions. Celui ci permet d'affiner les critères de sélection.

Nous verrons des exemples dans la seconde partie de ce document.

L'exemple suivant renvoie un objet range contenant toutes les cellules vides de la plage :

```

Dim MaPlage As Range
Set MaPlage = Range(Cells(1, 1), Cells(10,
10)).SpecialCells(xlCellTypeBlanks)

```

## TextToColumns

Permet de redistribuer sur plusieurs cellules une cellule contenant du texte avec des séparateurs, similaire à la méthode OpenText de la collection WorkBooks.

## Plages particulières

## Ligne ou colonne entière

Il y a plusieurs méthodes pour sélectionner des lignes complètes. On peut utiliser la propriété EntireRow d'un objet Range. Par exemple :  
Range(Cells(1,1),Cells(2,1)).EntireRow renvoie les lignes 1 et 2.  
Sinon on peut utiliser la collection Rows de l'objet Range.  
MaFeuille.Range(Rows(1), Rows(2)) renvoie les mêmes lignes.

## UsedRange

Renvoie un objet Range qui représente l'ensemble de la plage utilisée dans la feuille. Attention toutefois, l'effacement du contenu d'une cellule la laisse comme étant utilisée dans la feuille jusqu'à un nouvel appel de la méthode UsedRange. Nous verrons ce point dans la deuxième partie de ce document.

## Plage nommée

Toute plage d'une feuille peut avoir un nom (propriété Name). Le fait de nommer une plage permet de pouvoir faire référence à celle-ci de façon simple, et de rendre la programmation indépendante de l'adresse réelle de la plage. Ces plages peuvent être discontinues. Sauf déclaration particulière, une plage nommée appartient à l'objet Workbook. Il y a là un danger lors de la duplication d'une feuille. Au moment de la duplication, toutes les plages nommées qui réfèrent à la feuille que l'on va dupliquer deviennent des plages propres à cette feuille (c'est à dire dont le nom de la plage se transforme de "NomPlage" en "NomFeuille!NomPlage", puis les noms sont dupliqués dans la nouvelle feuille. Notons que les noms spécifiques de feuilles apparaissent dans la collection Names de l'objet classeur, mais que seuls les noms spécifiques apparaissent dans la collection Names de l'objet feuille. Dans un classeur modèle, il est fortement conseillé d'utiliser au maximum les plages nommées.

## Objets graphiques (Chart & ChartObject)

Les objets graphiques servent à tracer des courbes. Il existe des feuilles graphiques (Chart) qui appartiennent à l'objet Workbook, et des graphiques incorporés (ChartObject) qui appartiennent à l'objet Worksheet. Bien que ces objets soient similaires, ils présentent quelques différences d'emploi. Je ne vais pas étudier l'ensemble des propriétés et méthodes de mise en forme de ces objets car elles sont assez faciles à utiliser. Par contre, nous allons regarder les objets constituants de l'objet Chart.

## Collection Charts & ChartObjects

Pour la collection Charts, la position relative des onglets donne le numéro d'index de l'objet, il n'est en général pas le même que l'index dans la collection Sheets. Pour la collection ChartObjects, le numéro d'index correspond à l'ordre de création.

Les propriétés / méthodes sont les mêmes que pour la collection Worksheets. Il y a juste une différence pour la méthode Add de la collection ChartObjects puisqu'on peut donner la position et les dimensions de l'objet.  
ChartObjects.Add(Left, Top, Width, Height)

Les coordonnées sont données en point.

## Evènements

### Gestion d'événement pour l'objet ChartObject

L'objet ChartObject étant contenu dans l'objet Worksheet, il ne possède pas de module objet qui lui soit propre. Pour pouvoir utiliser ces événements il faut déclarer l'objet "WithEvents".

Par exemple, dans le module de la feuille :

```
Private WithEvents MonGraphe As Graph
```

```
Et ensuite
```

```
Private Sub MonGraphe_BeforeDoubleClick(ByVal ElementID As Long, ByVal Arg1 As Long, ByVal Arg2 As Long, Cancel As Boolean)
```

```
.....
```

```
End Sub
```

Il faudra bien sûr faire l'affectation dans le code avec par exemple

```
Set MaFeuille.MonGraphe=MaFeuille.ChartObjects.Add(10,10,100,100)
```

Si on doit gérer les événements de nombreux graphiques incorporés, il convient d'écrire une classe pour cette gestion.

***BeforeDoubleClick / BeforeRightClick***

Private Sub MonGraphe\_BeforeDoubleClick(ByVal ElementID As Long, ByVal Arg1 As Long, ByVal Arg2 As Long, Cancel As Boolean)  
ElementID renvoie l'élément du graphique sur lequel le double click a eu une action, le contenu de Arg1 et Arg2 dépendent de cet élément.

***Calculate***

Se produit après l'ajout ou la modification d'une série.

***MouseDown, MouseUp, MouseMove***

Private Sub MonGraphe\_MouseDown(ByVal Button As Long, ByVal Shift As Long, ByVal X As Long, ByVal Y As Long)  
Private Sub MonGraphe\_MouseUp(ByVal Button As Long, ByVal Shift As Long, ByVal X As Long, ByVal Y As Long)  
Private Sub MonGraphe\_MouseMove(ByVal X As Long, ByVal Y As Long)  
Evènements de gestion de la souris (classique en Visual Basic).

***SeriesChange***

Private Sub MonGraphe\_SeriesChange(ByVal SeriesIndex As Long, ByVal PointIndex As Long)  
Les arguments renvoient la série sélectionnée et le point sélectionné.  
Cet événement se produit lorsque la valeur d'un point change.

## ***Propriétés et méthodes***

**CopyPicture**

MonGraphe.CopyPicture(Appearance, Format, Size)  
Généralement utilisée en Visual Basic, cette méthode permet de copier l'image d'un graphique, ce qui permet de le rendre indépendant de ses données. Attention, il n'est pas possible après de faire la procédure inverse.

**Export**

MonGraphe.Export(FileName, FilterName, Interactive)  
Sert à exporter le graphique dans un fichier image.

**GetChartElement**

MonGraphe.GetChartElement(X, Y, ElementID, Arg1, Arg2)  
Cette méthode s'utilise avec les évènements souris. En passant à cette méthode les arguments X et Y, il renvoie le type d'élément et des informations complémentaires dans ElementID, arg1 et arg2

**Location**

Permet de changer un objet Chart en ChartObject ou inversement.

**SetSourceData**

Permet de définir une plage contenant les données à tracer. C'est une des méthodes permettant de tracer des séries. Nous y reviendrons en détail dans la deuxième partie.

**DisplayBlanksAs**

Cette propriété définit comment sont gérées les cellules vides de la plage de données.

**HasAxis, HasLegend, HasTitle**

Doivent être misent à "True" pour pouvoir utiliser les objets axes, légende....

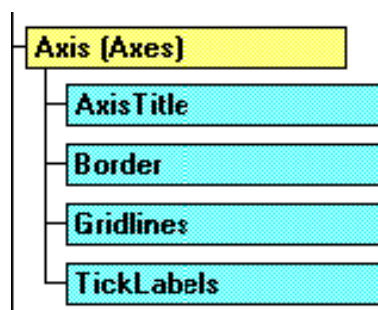
## ***Les objets constituants***

La programmation des graphiques se fait surtout en manipulant les objets constituants de celui-ci. Ces objets sont directement accessibles dans le cas d'un objet Chart, mais il faut passer par la propriété Chart pour y

accéder depuis un ChartObject (voir dans l'exemple pour "Legend " ci-dessous).

### Axis

Collection des axes du graphique. Pour identifier un seul axe, utiliser la méthode Axes (Type, Group) de l'objet Chart. Les valeurs pour Type sont xlCategory pour les abscisses et xlValue pour les ordonnées. Le modèle objet Axis est le suivant :

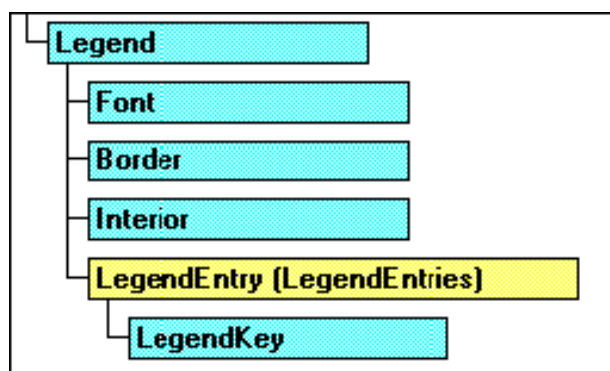


### ChartAreas, PlotAreas

Définissent des zones du graphique. Utilisées uniquement pour la mise en forme.

### Legend

Définit la légende du graphique. Le modèle objet est le suivant :



A chaque série du graphique correspond normalement un objet LegendEntry. Il est possible de supprimer un de ces objets, mais pas d'en modifier le texte directement (pour cela il faut modifier la propriété Name de la série). Par exemple pour supprimer la légende de la quatrième série d'un graphique incorporé :

```

Dim MonGraphe As ChartObject
Set MonGraphe = ActiveSheet.ChartObjects(1)
MonGraphe.Chart.Legend.LegendEntries(4).Delete
  
```

Comme je vous l'ai dit, vous voyez ici que je passe par la propriété Chart pour accéder à la légende.

### Series, SeriesCollection

C'est donc l'objet principal de la programmation des graphes, aussi allons nous regarder un peu plus en détail.

La collection SeriesCollection représente donc toutes les séries du graphique. On peut donc ajouter une série à l'aide de la méthode Add.

```

MonGraphe.SeriesCollection.Add(Source, Rowcol, SeriesLabels,
CategoryLabels, Replace)
  
```

L'argument Source contient l'objet Range contenant les données. On peut aussi ajouter une série vide à l'aide de la méthode NewSerie. Enfin on peut ajouter des données à une série à l'aide de la méthode Extend. Je reviendrai dans la deuxième partie de ce document sur la création dynamique de séries. On utilise directement l'objet Series pour sa mise en forme.

## Point, Points

La collection points représente tous les points d'une série, l'objet point un seul de ces points. Sachez juste qu'on ne peut pas ajouter un point à la collection Points.

## Conseils généraux

### Un peu de détente

Après ce parcours un peu fastidieux du modèle objet d'Excel, je vais un peu gloser sur les défauts habituels que l'on rencontre souvent en VBA. On trouve cinq grandes familles de "développeurs" VBA-Excel.

#### *L'enregistreur*

C'est la famille la mieux représentée, car la méthode est simple. On lance l'enregistrement automatique pour générer le maximum de code possible, puis on tente d'écrire le liant pour que l'ensemble fonctionne. Ce genre de code est facile à identifier car les variables ne sont pas déclarées et encore moins typées, le code est extrêmement lent, et une fois sur deux, la mise à jour de l'écran n'étant pas désactivée, le déplacement rapide de la sélection peut rendre une taupe épileptique.

#### *Le codeur fou*

En général il possède une bonne connaissance de la syntaxe Excel. Le code est même souvent très propre. Seulement comme il aime bien écrire des programmes, il se garde bien d'utiliser un seul raccourci pour alléger le code. Ainsi pour remplir quatre cases d'un formulaire, il en profitera pour générer l'ensemble du formulaire à chaque exécution au lieu d'utiliser un modèle.

#### *Le mathématicien*

Celui là ne pense pas qu'Excel puisse servir à autre chose que pour faire de la mise en page de tableau (la preuve, cela s'appelle un tableur). Aussi ne va-t-il pas confier un calcul à un logiciel, alors qu'il peut les faire dans le code.

#### *Le navigateur*

De la famille du grand Surfeur, il parcourt le Web à la recherche de bout de code qu'il pourrait copier pour mettre dans son programme. Cela crée un code un peu disparate, avec des parties bien écrites et d'autres moins. En général, ce genre d'application ne fait jamais bien ce qu'on lui demande, mais en échange, elle fait des tas de choses dont on n'a pas besoin.

#### *Le designer*

Cette dernière espèce est très fréquente en VB. Enthousiasmé par la conception de formulaire, on y trouve des boîtes de dialogue de toute beauté. Bien que la plupart des contrôles n'ait aucune utilité, reconnaissons que la beauté de l'oeuvre peut laisser rêveur.

## Les règles de bases

### *Poser le problème*

L'écriture d'une application quel que soit le langage utilisé doit toujours commencer par une analyse de ce que l'on veut faire. En général pour de la programmation Excel, on dispose d'une feuille Modèle (Template) dans laquelle on va amener des données pour obtenir des résultats (calculs et/ou graphiques). Ces données sont soit tirées d'un fichier, soit demandées à l'utilisateur. Dans le premier cas, l'interaction avec l'utilisateur devra être minimisée voire interdite, dans le deuxième on écrira une application événementielle.

### *Les variables*

Penser toujours à déclarer vos variables avant de les utiliser et donnez leur un type le plus précis possible.

Faite attention à préciser le type après chaque variable, en VBA

Dim Var1, Var2 As Integer est différent de Dim Var1 As Integer, Var2 As Integer.

Minimisez toujours la portée de vos variables, afin qu'elles n'existent que là où elles sont utiles.

Libérer la mémoire dès que possible, en utilisant des tableaux dynamiques plutôt que fixes, libérer aussi les objets.

Mettez dans des variables les propriétés dont vous avez besoin dans les boucles.

### ***Les fonctions***

N'abusez pas des fonctions, n'appellez pas de fonction dans les boucles. En général, passez des paramètres et n'utilisez pas des variables publiques. N'oubliez pas qu'une fonction peut modifier les paramètres passés ByRef et donc retourner des valeurs par ses arguments. Demandez-vous toujours ce que vous avez besoin de récupérer, ne récupérer pas un objet lorsqu'une adresse suffirait.

### ***Les objets***

Utilisez des blocs With lorsque vous avez besoin d'accéder à plusieurs propriétés / méthodes d'un objet. Mettez dans des variables les objets que vous appelez fréquemment. Faites attention, certaines propriétés / méthodes d'un objet renvoient un objet.

### ***L'application***

Pensez à désactiver le mode de calcul et la mise à jour d'écran. Remettez toujours l'objet Application dans l'état où il était. Désactivez les événements quand vous n'en avez pas besoin.

### ***Sélection***

Sauf lorsque c'est indispensable, n'utilisez pas les méthodes Select et Activate. Lorsque vous avez plusieurs objets feuilles ou classeurs, utilisez des variables pour les nommer, n'utilisez pas l'objet "Sélection".

### **Utilisation d'un modèle**

Pour simplifier la programmation, on utilise en général un modèle. Il est souvent inutile de programmer toute la mise en forme d'un document, alors que l'on peut appeler un modèle contenant déjà :

### **La mise en page**

Celle ci doit être relativement simple. Evitez le plus souvent de fusionner les cellules, ne surchargez pas vos feuilles de motifs et de bordures. Lorsqu'il s'agit d'un formulaire avec lequel l'utilisateur doit interagir, chercher à faire un formulaire fonctionnel et non une oeuvre d'art.

### **Les formules de calculs**

Dans vos formules, utilisez des plages nommées si vous êtes amené à supprimer / insérer des cellules. Faites attention aux références (relatives / absolues).

### **Les graphes**

Il est possible de placer déjà des graphes mis en forme dans le classeur. Une bonne solution consiste à utiliser pour les séries des plages nommées. Ainsi, le fait de nommer la plage lors de l'exécution remplira automatiquement le graphique.

### **Les plages nommées**

Nommez les plages spécifiques de la feuille à la création, sauf dans le cas où vous ne connaissez pas leur taille. Utilisez des noms clairs et significatifs.

## **Techniques de programmation**

### ***Évènements***

#### **Utiliser l'argument Target**

De nombreux événements Excel passe un argument Target qui est un objet Range contenant la plage concernée par l'événement. Cette plage peut être discontinue.

#### ***Intersection d'une plage avec Target***

Dans l'exemple suivant, la fonction met une croix (un X) lorsque l'on clique sur une cellule de la plage Nommée AjoutX

```
Private Sub Worksheet_SelectionChange(ByVal Target As Excel.Range)  
Dim MaPlage As Range
```

```
Set MaPlage = Application.Intersect(Range("AjoutX"), Target)
If Not MaPlage Is Nothing Then MaPlage.Value = "X"
End Sub
```

### Action sur une plage défini par Target

L'exemple suivant déclenche le calcul sur toutes les cellules dépendantes de la plage modifiée.

```
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
Dim MaPlage As Range
On Error Resume Next
Set MaPlage = Target.Dependents
Err.Clear
If Not MaPlage Is Nothing Then
MaPlage.Calculate
End If
End Sub
```

### Désactiver les évènements

Cela se fait avec Application.EnableEvents=False. Ceci est particulièrement utile pour les évènements Change et SelectionChange qui peuvent facilement créer un événement en cascade. Par exemple le code suivant est un événement en cascade.

```
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
ActiveCell.Offset(1).Value = 12
End Sub
```

Pour une exécution correcte, il faut écrire :

```
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
Application.EnableEvents = False
ActiveCell.Offset(1).Value = 12
End Sub
```

### Lire les informations d'un graphe

Dans un objet graphique, on peut utiliser les évènements de la souris pour savoir sur quelle partie du graphique se situe le pointeur. En elle-même la fonction suivante ne sert à rien, mais elle donnera une idée de la programmation à suivre. Cette fonction affiche dans la StatusBar les informations récupérées lors du déplacement de la souris.

```
Private Sub Chart_MouseMove(ByVal Button As Long, ByVal Shift As Long,
ByVal X As Long, ByVal Y As Long)
Dim MonElement As Long, Param1 As Long, param2 As Long
ActiveChart.GetChartElement X, Y, MonElement, Param1, param2
Select Case MonElement
Case xlChartArea
Application.StatusBar = "Zone de graphique"
Case xlChartTitle
Application.StatusBar = "Titre"
Case xlPlotArea
Application.StatusBar = "Zone de traçage"
Case xlLegend
Application.StatusBar = "Légende"
Case xlSeries
Application.StatusBar = "Série n° " & Param1 & "point n° " & param2
Case xlDataLabel
Application.StatusBar = "label de la série n° " & Param1 & "point n° "
& param2
Case xlLegendKey
Application.StatusBar = "légende de la série n° " & Param1
```

```
End Select
End Sub
```

## Application

N'oublions pas les fonctions de bases

Mise à jour d'écran Application.ScreenUpdating = False / True

Mode de calcul Application.Calculation= xlCalculationAutomatic / xlCalculationManual

Message d'alerte Application.DisplayAlerts= False / True

Vider le presse - papier Application.CutCopyMode= False

Bloquer les interactions souris-clavier Application.Interactive = False / True

## Récupérer les paramètres internationaux

On utilise pour cela la propriété "International". Attention, cette propriété est en **lecture seule**. Il est possible de modifier ces paramètres avec l'API Windows, mais pensez toujours alors à restaurer ces paramètres avant l'arrêt de votre application.

## Modifier les menus

Il est possible de supprimer des menus d'Excel lors de l'exécution, on utilise pour cela la collection CommandBars de l'objet Application. Pour trouver les références d'un menu, il suffit de connaître sa position. La barre de menu est CommandBars(1) les autres objets étant les barres d'outils. Il convient de faire preuve de cohérence. Lorsque l'on cherche à supprimer l'accès ou la visibilité d'un menu, on supprime de même son équivalent dans les barres d'outils et on bloque l'accès au menu "Outils-personnaliser".

Rappel : "Enabled" interdit l'accès, "Visible" masque l'élément

La fonction suivante désactive le menu "Insertion-Cellules", les boutons correspondant de la barre d'outils et le menu "Outils-personnaliser".

```
Public Sub DesacInsertDelCell()
With Application
With .CommandBars(1)
'Menu insertion cellule
.Controls(4).Controls(1).Enabled = False
'Menu outils personnaliser
.Controls(6).Controls(12).Enabled = False
End With
With Application.CommandBars("Formatting")
.Controls(15).Enabled = False
.Controls(16).Enabled = False
End With
End With
End Sub
```

## Classeur

### Ouvrir un classeur

*Selon un modèle*

```
Workbooks.add("MonFichier.xlt")
```

*N'ayant qu'une seule feuille*

```
Workbooks.add(xlWBATWorksheet)
```

*Avec un nombre fixe de feuille*

```
Application.SheetsInNewWorkbook=4
```

```
Workbooks.add
```

## Feuille



## Rendre les plages nommées spécifiques à leur feuilles

La macro suivante va affecter à chaque feuille les plages qui réfèrent à elle dans le classeur. Cette fonction n'a pas de véritable utilité dans un code, mais elle va nous permettre de voir deux concepts intéressants.

```
Public Sub AffectNom()
Dim MaFeuille As Worksheet, LeNom As Name, Recup As Range
For Each LeNom In ActiveWorkbook.Names
If InStr(1, LeNom.RefersTo, "!") > 0 And InStr(1, LeNom.Name, "!") = 0
Then
Set MaFeuille = Worksheets(Mid(LeNom.RefersTo, 2, InStr(1,
LeNom.RefersTo, "!") - 2))
Set Recup = Application.Evaluate(LeNom.RefersTo)
Recup.Name = MaFeuille.Name & "!" & LeNom.Name
LeNom.Delete
End If
Next
End Sub
```

La première remarque est de remarquer que l'on ne peut pas renommer une plage pour la rendre spécifique.

Si j'avais utilisé la ligne `LeNom.Name = MaFeuille.Name & "!" & LeNom.Name` cela n'aurait pas eu d'effet. Il est possible de changer le Nom pour un autre nom, mais pas par un nom spécifique.

La deuxième astuce est l'utilisation d'`Evaluate`. Cette fonction se retrouve souvent en VBA car elle permet de combler de nombreuses lacunes de la programmation Excel. Dans le cas présent, je l'utilise pour convertir une adresse en un objet Range.

N.B: Dans ce cas, `Evaluate` n'est pas nécessaire, la commande suivante fonctionne aussi :

```
Set Recup = Range(LeNom.RefersTo)
```

## Gestion des erreurs

Celle-ci se pratique de la même façon que dans Visual Basic. Pour une raison que je n'ai toujours pas comprise, beaucoup de développeurs VB la méprise, comme si son utilisation sous-entendait que le programmeur va faire des erreurs. Sachez toutefois qu'elle est dans certains cas extrêmement rapide, et qu'il est dommage de ne pas l'utiliser au nom d'une dogmatique tout à fait discutable. Prenons l'exemple suivant. Dans mon classeur je cherche à savoir si la feuille "pilotage" existe. Je peux évidemment parcourir la collection des feuilles pour la chercher, mais cela est assez lent. Le mieux est alors de faire :

```
Public Sub rempliTab()
Dim MaFeuille As Worksheet, Maplage As Range, compteur As Long
On Error Resume Next
Set MaFeuille = ActiveWorkbook.Worksheets("pilotage")
If Err.Number <> 0 Then
ActiveWorkbook.Worksheets.Add
After:=ActiveWorkbook.Worksheets(ActiveWorkbook.Worksheets.Count)
ActiveSheet.Name = "pilotage"
Set MaFeuille = ActiveSheet
Err.Clear
End If
On Error GoTo 0
Set Maplage = Worksheets("pilotage").Range(Cells(1, 1), Cells(20, 5))
For compteur = 1 To Maplage.Cells.Count
Maplage.Cells(compteur).Value = compteur
Next compteur
End Sub
```

## Plage

### Références dans les plages

Dans une plage définie, le système de référence est le même que pour une feuille, à savoir

MaPlage.Cells(Ligne,Colonne). La cellule supérieure gauche au sein de la plage prend l'adresse (1,1) et la cellule inférieure droite la valeur MaPlage.Cells(MaPlage.Rows.Count, MaPlage.Columns.Count).

Regardons le cas suivant :

```
Public Sub rempliTab()
Dim MaPlage As Range, compteur As Long
Set MaPlage = Worksheets("pilotage").Range(Cells(1, 2), Cells(20, 6))
For compteur = 1 To MaPlage.Cells.Count
MaPlage.Cells(compteur).Value = compteur
Next compteur
MaPlage.Range(Cells(3, 3), Cells(4, 4)).Interior.ColorIndex=3
MaPlage.Cells(1, 0).Interior.ColorIndex=4
End Sub
```

La référence MaPlage.Range(Cells(3,3),Cells(4,4)) renvoie une plage de quatre cellules dans le système d'adresse de la plage, c'est à dire dans notre exemple égale à ActiveSheet. Range(Cells(3,4),Cells(4,5)), puisque la plage commence à la colonne 2. Nous voyons à la ligne suivante que nous pouvons donner une adresse extérieure à la plage. Dans notre cas le numéro de colonne 0 représente la colonne à gauche de la plage. Ceci n'est pas sans inconvénients, car il n'y a pas de contrôle sur la présence d'une cellule dans la plage.

Donc en général l'adresse d'une cellule de l'objet Range se donne par Cells(Ligne, Colonne). Néanmoins dans une plage continue, on peut utiliser juste le numéro d'index de la cellule au sein de la collection. Excel donne ce numéro toujours dans l'ordre "à droite puis en bas". Ainsi le code ci dessus affichera "1" dans la cellule B1, "5" dans la cellule F1, "6" dans la cellule B2 etc....

Cette notation est très peu utilisée, mais elle peut permettre de parcourir une plage avec une seule boucle.

Pour passer d'un index à une adresse on emploie les formules suivantes :

Index Ligne, colonne

MaPlage.Cells(MonIndex\ MaPlage.Columns.Count+1, MonIndex mod MaPlage.Columns.Count)

Ligne, colonne Index

MonIndex=(Cellule.Row-1)\*MaPlage.Columns.Count+Cellule.Column

### Tableau contenant les valeurs ou les formules

Deux propriétés d'une plage peuvent être récupérées dans un tableau à partir d'une plage Value et Formula. L'avantage de cette méthode est qu'il est beaucoup plus rapide de parcourir un tel tableau que d'énumérer une grande collection de cellules. Dans le chapitre qui vient, nous allons regarder en détail le calcul sur les plages. Pour pouvoir faire quelques tests de performance, je vais utiliser l'API "GetTickCount" qui renvoie le nombre de millisecondes écoulées depuis le démarrage de ma session Windows. Mon code va donc se présenter ainsi :

```
Option Explicit
Private Declare Function GetTickCount Lib "kernel32" () As Long
Public Sub TestEff()
Dim MaPlage As Range, Depart As Long
Application.Calculation = xlCalculationManual
Depart = GetTickCount
Set MaPlage = Range(Cells(1, 1), Cells(25000, 10))
.. 'code à tester
MaPlage.Calculate
MsgBox GetTickCount - Depart
End Sub
```

N.B : "Option Explicit" oblige la déclaration des variables.

Pour créer ma plage de tests je vais utiliser la formule Excel "=Ligne()\*Colonne()" qui dans chaque cellule multiplier le numéro de la ligne par celui de la colonne. Ma ligne de remplissage sera :

MaPlage.FormulaLocal = "=Ligne()\*Colonne()"

Dans ce cas j'utilise FormulaLocal pour lui passer une formule appelant des fonctions en français, ce qui est strictement équivalent à utiliser :

MaPlage.Formula = "=ROW()\*COLUMN()"

En supposant par contre que la cellule A1 contienne déjà la formule ci-dessus, je pourrais recopier cette formule avec AutoFill.

MaPlage.Cells(1).AutoFill Destination:=MaPlage

Cependant le code suivant engendrera une erreur car AutoFill ne peut pas recopier une cellule dans deux directions simultanément. Je devrais donc utiliser :

```
MaPlage.Cells(1).AutoFill Destination:=MaPlage.Rows(1)
```

```
MaPlage.Rows(1).AutoFill Destination:=MaPlage
```

Cette méthode est un peu plus lente que de faire :

```
MaPlage.Formula = MaPlage.Cells(1).Formula
```

Mais ayez déjà à l'esprit que le temps de remplissage de la plage représente environ 10% du temps de la fonction (le reste étant le temps de calcul).

#### ***A ne jamais faire***

Une autre méthode pourrait être de parcourir la collection des cellules et d'écrire :

```
For Each MaCellule In MaPlage.Cells
```

```
MaCellule.FormulaLocal = "=Ligne()*Colonne()"
```

```
Next
```

Seulement dans ce cas, le temps de traitement est multiplié par **dix**.

#### ***Utiliser un tableau (variant)***

Comme je vous l'ai dit au début on peut affecter les valeurs d'un tableau à la propriété "Value" ou "Formula" d'une plage. Je déclare mon tableau comme un Variant car certaines versions d'Excel n'acceptent pas les tableaux typés du fait que le tableau peut contenir du texte. Regardons les trois codes suivants:

```
Public Sub TestEff()  
Dim MaPlage As Range, Depart As Long  
Depart = GetTickCount  
Application.Calculation = xlCalculationManual  
Set MaPlage = Range(Cells(1, 1), Cells(25000, 10))  
MaPlage.FormulaLocal = "=Ligne()*Colonne()"  
MaPlage.Calculate  
MsgBox GetTickCount - Depart  
End Sub  
  
Public Sub TestEff()  
Dim MaPlage As Range, Depart As Long  
Dim MonTab As Variant, comptLig As Long, comptCol As Long  
Depart = GetTickCount  
Application.Calculation = xlCalculationManual  
Set MaPlage = Range(Cells(1, 1), Cells(25000, 10))  
ReDim MonTab(1 To MaPlage.Rows.Count, 1 To MaPlage.Columns.Count)  
For comptLig = 1 To MaPlage.Rows.Count  
For comptCol = 1 To MaPlage.Columns.Count  
MonTab(comptLig, comptCol) = "=Ligne()*Colonne()"  
Next comptCol  
Next comptLig  
MaPlage.FormulaLocal = MonTab  
MsgBox GetTickCount - Depart  
End Sub  
  
Public Sub TestEff()  
Dim MaPlage As Range, Depart As Long  
Dim MonTab As Variant, comptLig As Long, comptCol As Long  
Depart = GetTickCount  
Application.Calculation = xlCalculationManual  
Set MaPlage = Range(Cells(1, 1), Cells(25000, 10))  
ReDim MonTab(1 To MaPlage.Rows.Count, 1 To MaPlage.Columns.Count)  
For comptLig = 1 To MaPlage.Rows.Count  
For comptCol = 1 To MaPlage.Columns.Count  
MonTab(comptLig, comptCol) = comptLig * comptCol  
Next comptCol  
Next comptLig  
MaPlage.Value = MonTab  
MsgBox GetTickCount - Depart  
End Sub
```

Le premier et le deuxième code font exactement la même chose, c'est à dire qu'ils mettent la formule de calcul dans chaque cellule de la plage, mais le premier le fait cent fois plus vite. Il convient donc de ne pas utiliser de tableau Variant sur les formules mais bien uniquement sur les valeurs.

Le premier et le troisième code donnent les mêmes résultats, mais une plage contient des formules, et l'autre uniquement des valeurs. Par contre, la vitesse de traitement est strictement identique. Comme cette plage va me servir de plage de valeur je peux donc utiliser indifféremment une des deux fonctions suivantes (la première étant un peu plus rapide) :

```
Public Sub TestEff()
Dim MaPlage As Range
Dim MonTab As Variant, comptLig As Long, comptCol As Long
Set MaPlage = Range(Cells(1, 1), Cells(25000, 10))
ReDim MonTab(1 To MaPlage.Rows.Count, 1 To MaPlage.Columns.Count)
For comptLig = 1 To MaPlage.Rows.Count
For comptCol = 1 To MaPlage.Columns.Count
MonTab(comptLig, comptCol) = comptLig * comptCol
Next comptCol
Next comptLig
MaPlage.Value = MonTab
End Sub
Public Sub TestEff()
Dim MaPlage As Range
Set MaPlage = Range(Cells(1, 1), Cells(25000, 10))
MaPlage.FormulaLocal = "=Ligne()*Colonne()"
MaPlage.Value = MaPlage.Value
End Sub
```

A noter que la ligne `MaPlage.Value = MaPlage.Value` remplace les formules de la plage par les valeurs (c'est l'équivalent d'un collage spécial valeur).

Nous allons continuer à évaluer l'utilisation de ces tableaux en faisant l'exercice suivant. Je pars donc de ma plage et je veux que chaque cellule contienne la valeur  $\text{Cellule} \times 2 + 3$ . Dans ma feuille, j'ai nommé ma plage précédente "PlageSource" (ce qui n'est pas original).

### *Calcul par la feuille*

La première méthode consiste à faire le calcul par la feuille. J'utilise donc le code suivant :

```
Public Sub TestEff()
Dim Depart As Long, MaPlage As Range
Depart = GetTickCount
Application.Calculation = xlCalculationManual
'méthode1 calcul par feuille
Set MaPlage = Range("PlageSource").Offset(0,
Range("PlageSource").Columns.Count)
MaPlage.FormulaLocal = "=LC(-" & Range("PlageSource").Columns.Count &
")*2+3"
MaPlage.Calculate
Range("PlageSource").Value = MaPlage.Value
MaPlage.Clear
Set MaPlage = Nothing
MsgBox GetTickCount - Depart
End Sub
```

C'est une utilisation standard des feuilles de calcul, mais il faut être sûr d'avoir la place pour créer une plage intermédiaire.

### *Calcul par collage spécial*

Cette méthode est beaucoup plus originale. Dans mon exemple elle n'est pas logique d'emploi, mais sachez qu'elle existe et qu'elle est aussi rapide que les autres.

```

Public Sub TestEff()
Dim Depart As Long, MaPlage As Range
Depart = GetTickCount
Application.Calculation = xlCalculationManual
Set MaPlage = Range("PlageSource").Offset(0,
Range("PlageSource").Columns.Count)
MaPlage.Value = 2
MaPlage.Copy
Range("PlageSource").PasteSpecial xlPasteValues,
xlPasteSpecialOperationMultiply
MaPlage.Value = 3
MaPlage.Copy
Range("PlageSource").PasteSpecial xlPasteValues,
xlPasteSpecialOperationAdd
MaPlage.Clear
Set MaPlage = Nothing
MsgBox GetTickCount - Depart
End Sub

```

Comme vous le voyez, je colle des plages de taille identique sur ma plage en lui faisant faire une opération à chaque fois. Notez que dans ces deux codes, je libère l'objet Maplage en fin de fonction ce qui n'est pas nécessaire puisque la variable va être détruite, mais ce qu'il ne faut pas oublier de faire dans une fonction plus longue.

#### ***Calcul par tableau variant***

Avec ce code je passe par un tableau en mémoire.

```

Public Sub TestEff()
Dim Depart As Long, MaPlage As Range, MonTab As Variant, comptLig As Long,
comptCol As Long
Depart = GetTickCount
MonTab = Range("PlageSource").Value
For comptLig = 1 To UBound(MonTab, 1)
For comptCol = 1 To UBound(MonTab, 2)
MonTab(comptLig, comptCol) = MonTab(comptLig, comptCol) * 2 + 3
Next comptCol
Next comptLig
Range("PlageSource").Value = MonTab
Erase MonTab
MsgBox GetTickCount - Depart
End Sub

```

#### ***Calcul par tableau avec Evaluate***

Enfin cette dernière méthode utilise la fonction "Evaluate"

```

Public Sub TestEff()
Dim Depart As Long, MaPlage As Range, MonTab As Variant
Depart = GetTickCount
Application.ReferenceStyle = xlA1
MonTab = Application.Evaluate(Range("plagesource").Address(True, True,
xlA1, True) & "* 2 + 3")
MaPlage.Value = MonTab
Erase MonTab
MsgBox GetTickCount - Depart
End Sub

```

Pour un bon fonctionnement de Evaluate, il faut que l'environnement Excel soit en mode A1.

**Synthèse**

Je vous donne ces différentes syntaxes car elles sont sensiblement équivalentes en vitesse. Selon les cas il convient d'en utiliser une plutôt que l'autre, mais c'est à vous de savoir.

**UsedRange et "SpecialCells(xlCellTypeLastCell)"**

Il y a un piège potentiel lors de la recherche de la dernière cellule de la feuille. En pratique Excel définit comme dernière cellule, la cellule de rang le plus élevé de la plage UsedRange. Regardons le code suivant

```
Range(Cells(1, 1), Cells(100, 10)).Value = "12"
Range(Cells(50, 1), Cells(100, 10)).Clear
MsgBox ActiveSheet.Cells.SpecialCells(xlCellTypeLastCell).Address
```

La boîte de dialogue va renvoyer l'adresse J100, pourtant la dernière cellule non vide est la cellule J49. Ceci vient du fait qu'Excel ne remet pas à jour automatiquement l'adresse de la dernière cellule lors d'un effacement. Pour forcer cette mise à jour, il faut faire appel **explicitement** à la propriété UsedRange. Dès lors le code correct est :

```
MsgBox ActiveSheet.UsedRange.SpecialCells(xlCellTypeLastCell).Address
```

N.B : L'appel de UsedRange déclenche toujours la mise à jour de la dernière cellule.

**A la recherche des cellules vides**

Voilà un grand classique de la programmation d'Excel. En fait plusieurs méthodes sont utilisables selon que

	1	2	3		1	2	3	
1	12	=2*LC(-1)	Base		1	12	24	Base
2	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		2	12	24	12
3		=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		3		0	12
4	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		4	12	24	
5	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		5	12	24	12
6		=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		6		0	12
7	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		7	12	24	
8	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		8	12	24	12
9	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		9	12	24	12
10		=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		10		0	12
11	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		11	12	24	
12	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		12	12	24	12
13		=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		13		0	12
14	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		14	12	24	
15	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		15	12	24	12
16	12	=2*LC(-1)	=SI(ESTVIDE(L(-1)C(-2));"";L(-1)C(-2))		16	12	24	12
17					17			
18					18			

l'on souhaite savoir s'il y a des cellules vides ou si on veut pouvoir y accéder.

Etudions le cas ci-dessus avec le code ci-dessous :

```
Public Sub RechercheVide()
Dim MaPlage As Range, NbVide As Integer
NbVide = Application.WorksheetFunction.CountBlank(Range(Cells(1, 1),
Cells(16, 3)))
If NbVide>0 Then Set MaPlage = Range(Cells(1, 1), Cells(16,
```

```
3)).SpecialCells(xlCellTypeBlanks)
MsgBox NbVide - MaPlage.Cells.Count
End Sub
```

La boîte de dialogue affiche "4". Cela vient du fait que l'ordre `SpecialCells(xlCellTypeBlanks)` cherche les cellules effectivement vides (c'est à dire qui ne contiennent ni valeur ni formule) alors que la fonction `CountBlank` compte les cellules dont la **valeur** est vide. On obtient donc deux informations différentes. Supposons maintenant que je le but soit d'obtenir un objet `Range` avec toutes les cellules dont la valeur est vide. Cette plage ne peut pas être trouvée directement, nous allons donc écrire la fonction qui le fait. J'utilise donc ma fonction `ChercheVide`, qui appelle une fonction `PlageCelluleVide` qui renvoie l'objet `Range` désiré.

```
Public Sub ChercheVide()
Dim MaPlage As Range, NbVide As Integer, MaFeuille As Worksheet
NbVide = Application.WorksheetFunction.CountBlank(Range(Cells(1, 1),
Cells(16, 3)))
If NbVide>0 Then Set MaPlage = PlageCelluleVide(Range(Cells(1, 1),
Cells(16, 3)))
MsgBox NbVide - MaPlage.Cells.Count
End Sub

Public Function PlageCelluleVide(PlageSource As Range) As Range
Dim MaPlage As Range, PremAdresse As String
Set MaPlage = PlageSource.Cells.Find("", PlageSource.Cells(1, 1),
xlValue, xlWhole)
If Not MaPlage Is Nothing Then
PremAdresse = MaPlage.Address
Set PlageCelluleVide = MaPlage
Do
Set MaPlage = PlageSource.FindNext(After:=MaPlage)
If MaPlage Is Nothing Then Set PlageCelluleVide = MaPlage Else
Set PlageCelluleVide = Union(PlageCelluleVide, MaPlage)
Loop While Not MaPlage Is Nothing And MaPlage.Address <>
PremAdresse
End If
End Function
```

Bien sûr on peut aussi faire une énumération de toutes les cellules, mais c'est beaucoup plus long. Vous noterez que quel que fut le cas, j'ai conditionné l'affectation de l'objet `Range` au fait qu'il y ait au moins une cellule vide afin de ne pas déclencher une erreur d'exécution.

### *Recherche de valeur particulière*

Ce genre de recherche suit le même principe que la recherche ci-dessus. La fonction de décompte s'appelle "CountIf", et on utilise de préférence la méthode "Find" à une énumération.

Il y a néanmoins une astuce très efficace bien que méconnue et qui consiste à utiliser la méthode `Autofilter`. Cette méthode ne fonctionne bien que sur une colonne mais rien n'empêche de faire une boucle. Je vais prendre un exemple. Je cherche l'ensemble des cellules dont la valeur vaut 24 dans une plage. Je devrais normalement utiliser `Find` avec le code suivant :

```
Public Sub Recherch1()
Dim MaPlage As Range, PlageCellule24 As Range, PremAdresse As String,
MaCell As Range
Set MaPlage = ActiveSheet.Cells(1, 15).CurrentRegion
Set MaCell = MaPlage.Cells.Find(24, MaPlage.Cells(1, 1), xlValue,
xlWhole)
If Not MaCell Is Nothing Then
PremAdresse = MaCell.Address
Set PlageCellule24 = MaCell
Do
Set MaCell = MaPlage.FindNext(After:=MaCell)
Set PlageCellule24 = Union(PlageCellule24, MaCell)
```

```

Loop While Not MaCell Is Nothing And MaCell.Address <>
PremAdresse
End If
End Sub

```

Ce code fonctionne parfaitement bien, mais le code suivant fait la même chose **cent cinquante fois plus vite** sur une plage de 4000 éléments.

```

Public Sub Recherche2()
Dim MaPlage As Range, PlageCellule24 As Range, compteur As Long
Set MaPlage = ActiveSheet.Cells(1, 15).CurrentRegion
For compteur = 1 To MaPlage.Columns.Count
MaPlage.Columns(compteur).AutoFilter field:=1, Criteria1:=24
If PlageCellule24 Is Nothing Then Set PlageCellule24 =
MaPlage.Columns(compteur).SpecialCells(xlCellTypeVisible) Else Set
PlageCellule24 =
Application.Union(MaPlage.Columns(compteur).SpecialCells(xlCellTypeVisible)
, PlageCellule24)
MaPlage.AutoFilter
Next compteur
PlageCellule24.Select
End Sub

```

Comme vous le voyez sur chaque colonne de la plage, j'applique l'AutoFilter sur la colonne. Celui ci masque toutes les lignes qui ne correspondent pas au critère, et je récupère la plage des cellules non masquées avec `MaPlage.Columns(compteur).SpecialCells(xlCellTypeVisible)`

### **Constantes et formules**

L'autre utilisation fréquente de la méthode "SpecialCells" est pour la gestion des tableaux contenant des formules. En effet dans le cas classique d'utilisation d'un tableau, il y a un certain nombre de données fixes (nommées constantes) qui alimentent les formules de calcul. Si ces plages sont continues, il n'y a pas de problèmes majeurs pour intervenir dessus, mais ce n'est pas toujours le cas. La ligne suivante supprime tous les nombres constants présents sur une feuille, sans toucher aux formules:

```

ActiveSheet.UsedRange.CurrentRegion.SpecialCells(xlCellTypeConstants,
xlNumbers).ClearContents

```

L'exemple suivant verrouille toutes les cellules contenant une formule.

```

ActiveSheet.UsedRange.CurrentRegion.SpecialCells(xlCellTypeFormulas).Locked
=True

```

La ligne suivante renvoie une plage contenant toutes les cellules en erreur :

```

Set MaPlage = ActiveSheet.UsedRange.SpecialCells(xlCellTypeFormulas,
xlErrors)

```

### **Format conditionnel**

Je traite ce cas, car il concerne un sujet qui revient très souvent sur les forums Excel. Très souvent, on trouve des questions du style, "comment trouver les cellules dont le fond est rouge?". Comme les cellules d'Excel ne rougissent pas spontanément, c'est en général une recherche sur une cellule ayant un format conditionnel. Comme les règles d'un format conditionnel découlent du contenu d'une cellule (valeur ou formule) il convient de faire porter la recherche sur ce contenu, et non sur l'effet visuel qui n'est que le résultat d'un test de ce contenu. Néanmoins certains cas peuvent être plus complexes, nous allons donc regarder cela dans l'exemple suivant. Toutefois, ne perdons pas de vue que le but est **toujours** d'éviter une énumération des cellules. Envisageons le cas de la recherche de doublons sur une colonne.

#### ***L'objet FormatCondition***

Chaque cellule peut avoir une collection FormatConditions qui peut contenir jusqu'à trois objets FormatCondition. Cet objet se décompose globalement en deux parties, les règles et le format. Pour simplifier, on définit des règles qui, lorsqu'elles sont remplies, modifient le format de la cellule. Les propriétés de format s'obtiennent en passant par les objets borders, font et interior. Les règles se définissent à l'aide de la méthode Add de la façon suivante.



Add(Type, Operator, Formula1, Formula2)

Type est soit xlCellValue (valeur de la cellule) soit xlExpression (formule n'étant pas contenue dans la cellule)

Operator peut être xlBetween, xlEqual, xlGreater, xlGreaterEqual, xlLess, xlLessEqual, xlNotBetween ou xlNotEqual, il est ignoré si le type est xlExpression.

Formula1 est la valeur ou l'expression associée au format conditionnel. Il peut s'agir d'une valeur constante, d'une chaîne, d'une référence de cellule ou d'une formule.

Formula2 est la valeur ou l'expression associée au second élément du format conditionnel lorsque Operator vaut xlBetween ou xlNotBetween (sinon, l'argument est ignoré). Il peut s'agir d'une valeur constante, d'une chaîne, d'une référence de cellule ou d'une formule.

N.B : il existe une méthode Modify qui permet de changer une règle existante.

Pour reprendre le cas qui nous intéresse, je veux que le fond des cellules doublonnées soit rouge. J'utilise alors le code suivant :

```
With ActiveSheet.Columns(11).FormatConditions
.Add Type:=xlExpression, Formula1:="=NB.SI(C11;LC)>1"
.Item(.Count).Interior.ColorIndex = 3
End With
```

Dans ce cas, toutes les cellules doublonnées de la colonne 11 auront un fond rouge. C'est là que l'utilisateur se demande comment récupérer les cellules ayant un fond rouge et qu'une énumération de la plage lui paraît inévitable. Pourtant, il ne faut pas poser le problème ainsi. Le format conditionnel sert à donner une indication "visuelle" à l'utilisateur. Notons d'ailleurs que le format conditionnel est volatile, il n'est donc pas détecté comme un format différent lors d'une énumération de cellule. Lorsque l'on veut accéder à la plage des doublons on doit passer par un tableau, et utiliser le tri. Je vous donne le code ci-dessous

```
Sub PlageDoublon()
Dim Depart As Long, maplage As Range, TabRes() As Long, MaChaine As String
Dim montab As Variant, comptX As Long
Application.ScreenUpdating = False
Depart = GetTickCount
Application.Calculation = xlCalculationManual
Set maplage = ActiveSheet.Range(Cells(1, 11), Cells(1, 11).End(xlDown))
With maplage.Offset(, -1)
.FormulaLocal = "=ligne()"
.Value = .Value
End With
Set maplage = maplage.Offset(, -1).Resize(, 2)
maplage.Sort maplage.Cells(1, 2), xlAscending
montab = maplage.Value
ReDim TabRes(1 To 1)
For comptX = 2 To UBound(montab, 1) - 1
If montab(comptX, 2) = montab(comptX + 1, 2) Or montab(comptX, 2) =
montab(comptX - 1, 2) Then
TabRes(UBound(TabRes)) = montab(comptX, 1)
ReDim Preserve TabRes(1 To UBound(TabRes) + 1)
End If
Next comptX
Erase montab
maplage.Sort maplage.Cells(1, 1), xlAscending
maplage.Columns(1).ClearContents
Set maplage = maplage.Offset(, 1).Resize(, 1)
maplage.Cells.EntireRow.Hidden = True
If UBound(TabRes) > 1 Then
For comptX = 1 To UBound(TabRes) - 1
maplage.Cells(TabRes(comptX), 1).EntireRow.Hidden = False
Next comptX
```

```

End If
Set maplage = maplage.SpecialCells(xlCellTypeVisible)
maplage.Interior.ColorIndex = 3
maplage.Cells.EntireRow.Hidden = False
Application.Calculation = xlCalculationAutomatic
Application.ScreenUpdating = True
MsgBox GetTickCount - Depart
End Sub

```

Cette fonction est extrêmement rapide sur une très grande plage contenant de nombreux doublons (pour 65000 éléments contenant 4700 doublons elle prend environ 9 secondes contre plus de dix minutes pour une méthode "standard").

### **AdvancedFilter**

Si le but est simplement de récupérer une plage sans doublon, ce qui est différent de ce qu'on a cherché à faire au dessus on utilise alors un filtre particulier AdvancedFilter

Sa syntaxe est la suivante

expression.AdvancedFilter(Action, CriteriaRange, CopyToRange, Unique)

Action est soit xlFilterInPlace, soit xlFilterCopy. Dans le premier cas le filtre est dit "en place", il fonctionne alors comme les filtres classiques par masquage des colonnes. Dans l'autre cas, le résultat apparaît sur une plage différente et continue.

CriteriaRange est la plage contenant les critères du filtre, pour plus de renseignements sur son fonctionnement consultez dans l'aide la rubrique " Exemples de critères pour le filtre élaboré".

CopyToRange est la plage de destination si le filtre n'est pas en place.

Unique permet justement d'enlever les doublons.

Le code suivant crée donc une plage sans doublon à droite de la plage source.

```

Sub PlageSansDoublons()
Dim maplage As Range, Depart As Long
Application.ScreenUpdating = False
Application.Calculation = xlCalculationManual
Set maplage = ActiveSheet.Range(Cells(1, 1), Cells(1, 1).End(xlDown))
maplage.AdvancedFilter xlFilterCopy, , maplage.Offset(, 1).Resize(1,
1), True
Application.ScreenUpdating = True
Application.Calculation = xlCalculationAutomatic
End Sub

```

Notons que pour une plage de 65536 cellules contenant 4700 doublons le traitement prends tout de même environ 7 minutes. Un filtre en place serait d'ailleurs plus long. Il peut être rentable dès lors de créer sa propre fonction, dérivée de celle vu auparavant, afin d'accélérer notablement la vitesse de traitement.

### **Validation**

Notons que nous pouvons, dans le cas d'un formulaire, limiter les saisies à l'aide de l'objet validation. Dans notre cas le code suivant interdira la possibilité de saisir un doublon.

```

Public Sub SaisieDoublon()
Dim maplage As Range
Set maplage = ActiveSheet.Columns(11)
With maplage.Validation
.Add Type:=xlValidateCustom, AlertStyle:=xlValidAlertStop,
Formula1:="=NB.SI(C11;LC)=1"
.ErrorTitle = "valeur déjà existante"
End With
End Sub

```

## **Graphique**

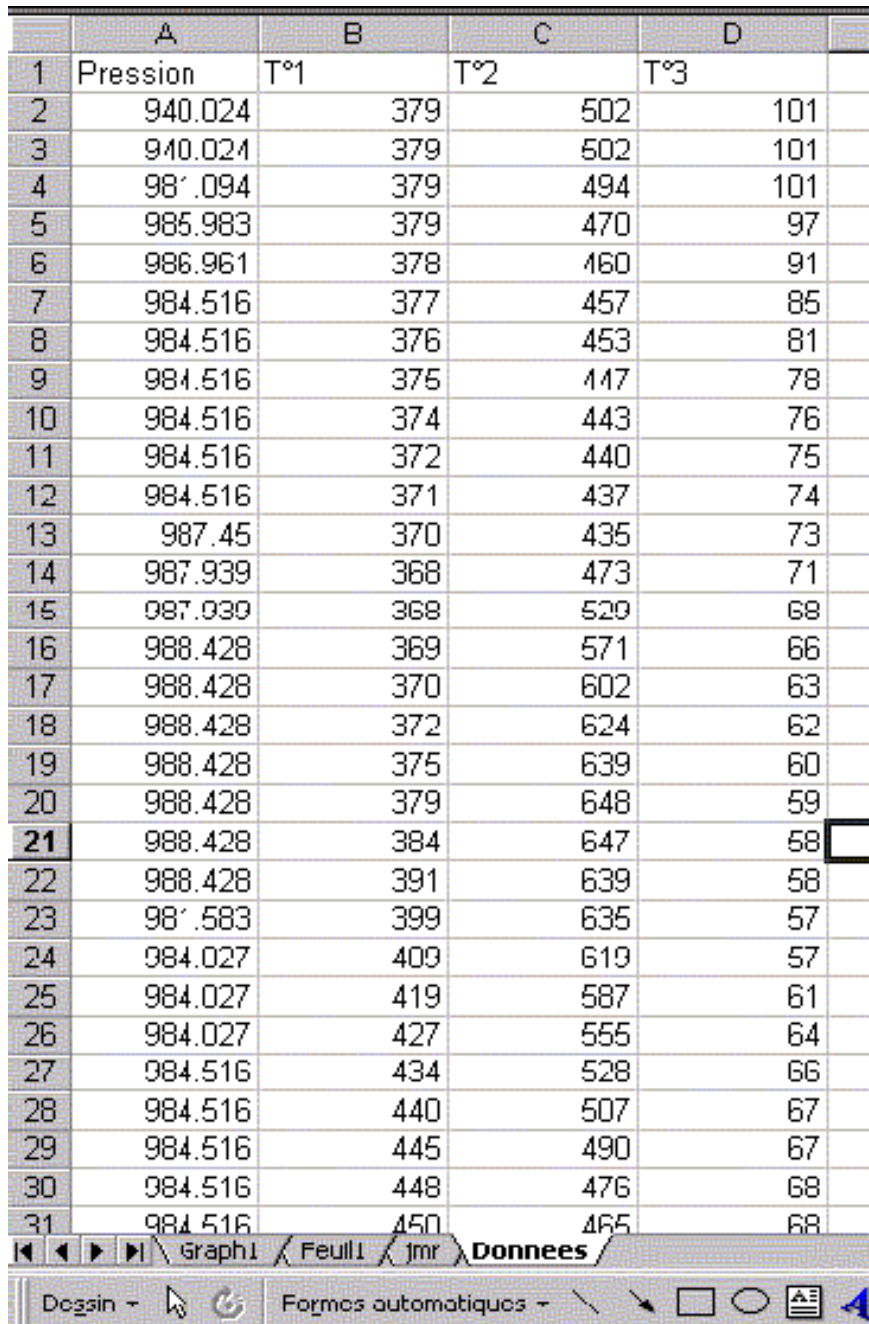
La mise en forme des graphiques par programme n'est pas très compliquée. Nous verrons un exemple de

mise en forme un peu complexe à la fin de ce chapitre. Le problème majeur de la manipulation des graphiques est la manipulation des séries.

### Création / ajout de séries

Il y a plusieurs méthodes pour ajouter des séries. La plus simple, mais aussi la moins souple et l'ajout implicite à la création. En effet si une plage de cellules est sélectionnée lors de la création du graphe, Excel trace les séries automatiquement. Sauf dans certains cas triviaux, cette méthode est à éviter. Pour regarder l'ajout de séries je vais prendre le cas du tracé nuage de points suivant :

	A	B	C	D	
1	Pression	T°1	T°2	T°3	
2	940.024	379	502	101	
3	940.024	379	502	101	
4	987.094	379	494	101	
5	985.983	379	470	97	
6	986.961	378	460	91	
7	984.516	377	457	85	
8	984.516	376	453	81	
9	984.516	375	447	78	
10	984.516	374	443	76	
11	984.516	372	440	75	
12	984.516	371	437	74	
13	987.45	370	435	73	
14	987.939	368	473	71	
15	987.930	368	520	68	
16	988.428	369	571	66	
17	988.428	370	602	63	
18	988.428	372	624	62	
19	988.428	375	639	60	
20	988.428	379	648	59	
21	988.428	384	647	58	
22	988.428	391	639	58	
23	987.583	399	635	57	
24	984.027	409	619	57	
25	984.027	419	587	61	
26	984.027	427	555	64	
27	984.516	434	528	66	
28	984.516	440	507	67	
29	984.516	445	490	67	
30	984.516	448	476	68	
31	984.516	450	465	68	



Soit trois colonnes de températures qui seront mes ordonnées, et Pression qui sera ma colonne d'abscisses. Si j'utilise le code suivant, je vais obtenir un tracé de quatre séries au lieu des trois séries que je désire.

```
Public Sub TestGraphe()
```

```
Dim MonGraphe As Chart
Range(Cells(1, 1), Cells(99, 4)).Select
ThisWorkbook.Charts.Add
End Sub
```

Bien entendu, je pourrais récupérer les valeurs de la première série et les mettre comme abscisse de chaque série, mais ce ne serait ni clair, ni efficace.

Attention, cela induit qu'Excel essaie toujours d'interpréter la sélection de la feuille active comme une plage de données. Il faut donc, si on a utilisé des sélections dans le code antérieur à la création du graphe, veiller à ramener la sélection à une seule cellule.

### ***Création par la propriété DataSource***

Lorsque la plage est bien positionnée, c'est à dire avec les abscisses en première ligne ou colonne, on peut utiliser cette méthode. Elle présente l'avantage d'implémenter un grand nombre de séries simultanément.

```
Public Sub TestGraphe()
Dim MonGraphe As Chart, MaPlage As Range
Set MaPlage = Worksheets("donnees").Range(Cells(1, 1), Cells(99, 4))
Set MonGraphe = ThisWorkbook.Charts.Add
MonGraphe.ChartType = xlXYScatter
MonGraphe.SetSourceData MaPlage, xlColumns
End Sub
```

### ***Création par ajout***

Cette méthode est similaire à la précédente, seulement on ajoute les données à la collection des séries.

```
Public Sub TestGraphe()
Dim MonGraphe As Chart, MaPlage As Range
Set MaPlage = Worksheets("donnees").Range(Cells(1, 1), Cells(99, 4))
Set MonGraphe = ThisWorkbook.Charts.Add
MonGraphe.ChartType = xlXYScatter
MonGraphe.SeriesCollection.Add MaPlage, xlColumns, True, True
End Sub
```

### ***Création par copier / coller***

Cette méthode est particulière. Elle consiste à créer l'objet graphe au préalable puis de copier la plage de données afin de réaliser un collage de séries.

```
Public Sub TestGraphe()
Dim MonGraphe As Chart, MaPlage As Range
Set MaPlage = Worksheets("donnees").Range(Cells(1, 1), Cells(99, 4))
Set MonGraphe = ThisWorkbook.Charts.Add
MonGraphe.ChartType = xlXYScatter
MaPlage.Copy
MonGraphe.SeriesCollection.Paste xlColumns, True, True, True, True
End Sub
```

Ces deux méthodes sont efficaces, mais utilisables uniquement dans certains cas. Nous allons voir maintenant des méthodes standards beaucoup plus universelles. Ces méthodes reposent sur un principe différent. On crée d'abord l'objet série, puis on lui affecte ses valeurs. Cela permet de travailler sur des plages discontinues, et la position de la colonne contenant les abscisses n'a plus d'importance

### ***Création par valeur***

```
Public Sub TestGraphe()
Dim MonGraphe As Chart, MaPlage As Range, MaSerie As Series, compteur
As Long
Set MaPlage = Worksheets("donnees").Range(Cells(2, 1), Cells(99, 4))
Set MonGraphe = ThisWorkbook.Charts.Add
MonGraphe.ChartType = xlXYScatter
For compteur = 2 To MaPlage.Columns.Count
Set MaSerie = MonGraphe.SeriesCollection.NewSeries
```

```

MaSerie.Values = "=" & MaPlage.Columns(compteur).Address(True,
True, xlR1C1, True)
MaSerie.XValues = "=" & MaPlage.Columns(1).Address(True, True,
xlR1C1, True)
MaSerie.Name = "=" & MaPlage.Cells(1).Offset(-1, compteur -
1).Address(True, True, xlR1C1, True)
Next compteur
End Sub

```

Comme nous le voyons, dans ce cas je passe par une création série par série de mon graphe.

### Création par formule

Similaire à la méthode précédente mais un peu moins lisible.

```

Public Sub TestGraphe()
Dim MonGraphe As Chart, MaPlage As Range, MaSerie As Series, compteur
As Long, toto
Set MaPlage = Worksheets("donnees").Range(Cells(2, 1), Cells(99, 4))
Set MonGraphe = ThisWorkbook.Charts.Add
MonGraphe.ChartType = xlXYScatter
For compteur = 2 To MaPlage.Columns.Count
Set MaSerie = MonGraphe.SeriesCollection.NewSeries
MaSerie.Formula = "=SERIES(" & MaPlage.Cells(1).Offset(-1, compteur
- 1).Address(True, True, xlR1C1, True) & "," &
MaPlage.Columns(1).Address(True, True, xlR1C1, True) & "," &
MaPlage.Columns(compteur).Address(True, True, xlR1C1, True) & "," &
compteur - 1 & ")"
Next compteur
End Sub

```

Pour mémoire une formule d'une série suit la syntaxe suivante :  
SERIES(Nom,PlageX,PlageY,Ordre)

### Modification de séries

La modification est en général de deux types : la modification d'une plage ou l'extension de la série. La modification de la plage se fait en attaquant une des propriétés Values, Xvalues ou Formula. L'extension se fait avec la méthode Extend. Je ne donnerai pas d'exemple dans ce cas puisque la méthode est rarement utilisée.

### Série de plages discontinues

Nous allons voir un cas particulier, qui peut être utile. Tout d'abord, il faut savoir qu'il n'y a pas obligation à utiliser des plages continues, ni même des plages de dimensions identiques pour les abscisses et les ordonnées. Néanmoins, créer par le code une série de plages discontinues n'est pas évident. Pour étudier ce cas je reprends mon exemple précédent, mais en cherchant à tracer un point sur 10. Le code suivant fait cela :

```

Public Sub TestGraphe1()
Dim MonGraphe As Chart, MaPlage As Range, compteur As Long, toto
Set MaPlage = Worksheets("Donnees").Cells(2, 1).Resize(, 4)
For compteur = 1 To Worksheets("Donnees").Cells(2,
1).Resize(Worksheets("Donnees").Cells(2, 1).End(xlDown).Row - 1).Rows.Count
\ 10
Set MaPlage = Application.Union(MaPlage, MaPlage.Offset(compteur *
10 + 1))
Next compteur
Set MonGraphe = ThisWorkbook.Charts.Add
MonGraphe.ChartType = xlXYScatter
MonGraphe.SetSourceData MaPlage, xlColumns
End Sub

```

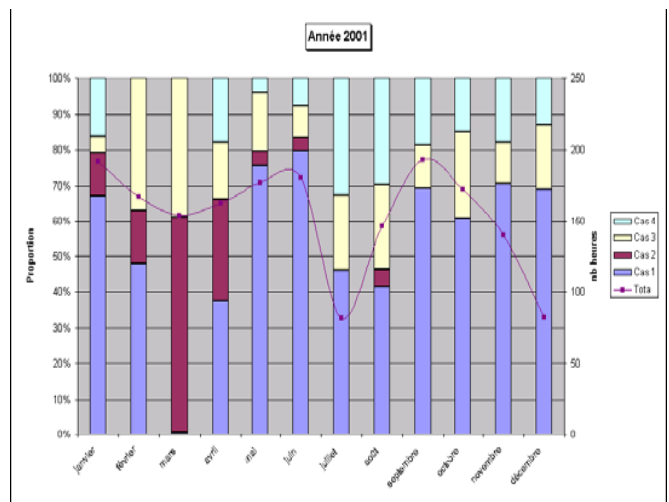
Pourtant ce code n'est pas utilisable à volonté, puisque passé un certain nombre de points, Excel ne peut

plus écrire la formule série à cause d'un trop grand nombre de caractères.

### Exemple 1 : création d'un graphique personnalisé

Dans ce premier exemple nous allons créer un graphique composé à partir d'un tableau pour obtenir par le code un graphe personnalisé (voir ci-dessous)

	7	8	9	10	11	12
Année 2001	Cas 1	Cas 2	Cas 3	Cas 4	Total	
janvier	128.75	23.3	8.5	31.5	192.05	
février	80	25	62	0	167	
mars	1	93	60	0	154	
avril	61	46.75	25.75	29	162.5	
mai	133.75	7	28.5	7.5	176.75	
juin	144	6.75	15.75	14	180.5	
juillet	37.5	0	17.5	26.75	81.75	
août	60.5	7	35.25	43.25	146	
septembre	133.75	0	23.5	36	193.25	
octobre	105	0	41.5	25.75	172.25	
novembre	98.5	0	16	25	139.5	
décembre	56.5	0	14.75	10.75	82	



Dans ce cas, le tableau est correctement ordonné, nous pouvons donc procéder à une création groupée des séries. Il s'agit d'un graphique superposé, c'est à dire possédant deux types de courbes, avec des histogrammes cumulés et une série en nuage de points. Plusieurs méthodes de création seraient possibles, mais dans ce cas, la plus simple consiste à créer un graphique "histogramme cumulé" avec toutes les séries, puis à agir sur la dernière série pour la transformer en nuage de points.

Comme nous l'avons vu précédemment la fonction commencera ainsi :

```
Public Sub CreationGraphel()  
Dim MonGraphe As Chart, MaPlage As Range  
Set MaPlage = Worksheets("donnees").Range(Cells(2, 7), Cells(14, 12))  
Set MonGraphe = ThisWorkbook.Charts.Add  
MonGraphe.ChartType = xlColumnStacked100  
MonGraphe.SetSourceData MaPlage, xlColumns
```

A ce point j'ai créé un graphe de base contenant toutes les séries. Je vais maintenant modifier la série "total" afin de pouvoir la mettre en nuage de points et l'affecter à l'axe secondaire des ordonnées.

```
With MonGraphe.SeriesCollection(5)  
.ChartType = xlXYScatterSmoothNoMarkers  
.AxisGroup = 2  
With .Border  
.Weight = xlMedium  
.LineStyle = xlAutomatic  
.ColorIndex = 4  
End With  
End With
```

Je vais maintenant mettre les libellés des axes et du titre

```
With MonGraphe  
.HasTitle = True  
With .ChartTitle  
.Characters.Text = "ANNEE 2001"  
.Shadow = True  
.Border.Weight = xlHairline  
End With  
With .Axes(xlValue, xlPrimary)  
.HasTitle = True  
.AxisTitle.Characters.Text = "Proportion"
```

```

End With
With .Axes(xlValue, xlSecondary)
.HasTitle = True
.AxisTitle.Characters.Text = "Total (hrs)"
End With
End With

```

Et j'ai obtenu avec une fonction très courte le graphe désiré.

### Exemple 2 : création d'un graphique complexe

Dans cet exemple nous allons très largement compliquer la tâche en créant un graphique à l'aide du tableau ci-dessous.

	6	7	8	9	10	11	12	13
19			<b>Resultat</b>	<b>Moyenne</b>	<b>s</b>	<b>M + s</b>	<b>M - s</b>	<b>libellé</b>
20		<i>Ech 1</i>	55.6	55.6	1.24	56.04	54.4	55.6 ± 1.24
21		<i>Ech 2</i>	43.5	44.4	0.84	45.24	43.6	44.4 ± 0.84
22		<i>Ech 3</i>	53.8	52.1	1.65	53.75	50.5	52.1 ± 1.65
23		<i>Ech 4</i>	53.7	51.8	1.51	53.31	50.3	51.8 ± 1.51
24		<i>Ech 5</i>	51.3	54.3	1.23	55.53	53.1	54.3 ± 1.23
25		<i>Ech 6</i>	51.5	54.5	1.64	56.14	52.9	54.5 ± 1.64

Ce style de graphe est assez souvent utilisé pour des essais scientifiques. Dans le cas présent on trace des résultats moyens pour une expérience avec leur écart type de reproductibilité. On trace ensuite, les résultats de la nouvelle série d'expérience, (les ronds rouges) ce qui permet de visualiser rapidement une série de résultats anormaux (ce qui est le cas dans cet exemple).

Dans ce cas je vais créer les séries les unes après les autres. En effet, pour faire ce graphique de type StockHLC il faut que les trois premières séries soit dans l'ordre suivant (maximum, minimum, moyenne) ce qui n'est pas le cas de mon tableau. Ma fonction commencera ainsi :

```

Public Sub CreationGraphe2()
Dim MonGraphe As Chart, maplage As Range, MaSerie As Series, compteur
As Long
Dim Mini As Single, Maxi As Single
Set maplage = Worksheets("donnees").Range(Cells(20, 7), Cells(25, 13))
Mini = Application.WorksheetFunction.Min(maplage.Columns(1),
maplage.Columns(2))
Mini = Int(Mini / (Int(Log(Mini) / Log(10)) * 10)) * Int(Log(Mini) /
Log(10)) * 10
Maxi = Application.WorksheetFunction.Max(maplage.Columns(1),
maplage.Columns(2))
Maxi = (Int(Maxi / (Int(Log(Maxi) / Log(10)) * 10)) + 1) *
Int(Log(Maxi) / Log(10)) * 10
Set MonGraphe = ThisWorkbook.Charts.Add
For compteur = 1 To 3
Set MaSerie = MonGraphe.SeriesCollection.NewSeries
MaSerie.Values = "=" & maplage.Columns(Choose(compteur, 5, 6,
3)).Address(True, True, xlR1C1, True)
MaSerie.XValues = "=" & maplage.Columns(1).Address(True, True,
xlR1C1, True)
MaSerie.Name = Choose(compteur, "max", "min", "moyenne")
Next compteur
MonGraphe.ChartType = xlStockHLC

```

```

For Each MaSerie In MonGraphe.SeriesCollection
With MaSerie.Border
.LineStyle = xlContinuous
.Weight = xlThin
If MaSerie.PlotOrder = 3 Then .ColorIndex = 1 Else .ColorIndex
= 5
End With
Next

```

J'ajoute ensuite la série de résultats, que je mets en forme :

```

Set MaSerie = MonGraphe.SeriesCollection.NewSeries
With MaSerie
.ChartType = xlXYScatter
.Values = "=" & maplage.Columns(2).Address(True, True, xlR1C1,
True)
.XValues = "=" & maplage.Columns(1).Address(True, True, xlR1C1,
True)
.Name = "resultat"
.MarkerStyle = xlMarkerStyleCircle
.MarkerForegroundColorIndex = 3
.MarkerSize = 10
.ApplyDataLabels xlDataLabelsShowValue, False, True
With .DataLabels
.Position = xlLabelPositionLeft
.Font.ColorIndex = 3
End With
End With

```

Ensuite je vais faire une opération particulière, qui consiste à afficher les labels de la série "max" puis à modifier leur texte afin d'afficher une chaîne de la forme "Moyenne ± Ecart".

```

Set MaSerie = MonGraphe.SeriesCollection(1)
MaSerie.ApplyDataLabels xlDataLabelsShowLabel
For compteur = 1 To MaSerie.Points.Count
MaSerie.Points(compteur).DataLabel.Text = maplage(compteur,
7).Value
Next compteur

```

Enfin je supprime les deux entrées de la légende (min et max) et j'ajuste l'axe.

```

With ActiveChart.Axes(xlValue)
.MinimumScale = Mini
.MaximumScale = Maxi
End With
With MonGraphe.Legend
.LegendEntries(1).Delete
.LegendEntries(1).Delete
End With
End Sub

```

De voir apparaître deux fois LegendEntries(1).Delete peut sembler être une erreur, mais cela vient du fait que la suppression de la première entrée donne le rang un à la deuxième. Ceci est une source d'erreur, il convient donc de faire attention lors de l'utilisation de Delete sur les collections Excel.

## ***Piloter Excel avec Visual Basic 6***

Comme nous l'avons vu le pilotage est extrêmement simple puisque la programmation est la même. Il suffit juste d'ajouter Excel lors du typage des variables, et de déclarer tous les objets Excel utilisés comme variable. Notons toutefois que dans certains cas on peut s'affranchir de certains objets s'ils sont uniques ou inutiles dans la procédure. Par exemple, si mon traitement porte sur une seule feuille de calcul je pourrais utiliser la fonction suivante :



```
Private Sub TraiteExcel()  
Dim MonExcel As Excel.Application, MaFeuille As Excel.Worksheet  
Set MonExcel = New Excel.Application  
MonExcel.Workbooks.Add xlWBATWorksheet  
Set MaFeuille = MonExcel.ActiveSheet
```

Comme nous le voyons, je n'ai pas utilisé d'objet Workbook, puisque ma cible sera uniquement **une** feuille de calcul.

## Règles générales

### *Cohérence des références*

En général, on programme le pilotage en utilisant un seul style de référence (soit A1 soit L1C1). Il convient de n'utiliser qu'un seul style tout au long du programme et de mettre l'option sur ce style au démarrage. Ainsi si je travaille en mode L1C1, je devrais trouver au début de ma procédure :

```
MonExcel.ReferenceStyle = xlR1C1
```

N'oubliez pas que pour transformer les valeurs de colonnes, de lettres en nombres ou inversement, il suffit d'utiliser les propriétés `columns` ou `address`.

```
NumCol = MaFeuille.Columns("CE").Column
```

```
LetCol = MaFeuille.Columns(83).Address(True, False)
```

### *Quitter proprement*

En fin de procédure, et avant de fermer Excel, pensez toujours à détruire vos variables d'objets Excel. Ne pas le faire peut engendrer une erreur. Dans mon premier exemple une sortie correcte serait :

```
Set MaFeuille = Nothing  
MonExcel.Quit  
Set MonExcel = Nothing  
End Sub
```

De même méfiez-vous de la désactivation des messages d'alertes.

## Communiquer avec Excel

Le mode de communication le plus utilisé est la communication directe, c'est à dire la possibilité de lire ou d'écrire dans des objets Excel à partir du programme VB. Cette communication, se fait uniquement par l'intermédiaire des objets que le composant Excel fournit.

Il faut tout d'abord savoir qu'Excel est un serveur Out-of-process, c'est à dire qui ne s'exécute pas dans le même espace mémoire que votre application. Ceci fait que vous pouvez très bien le gérer de façon asynchrone, et lui faire exécuter des tâches pendant que votre propre application travaille. Pour pouvoir utiliser cette faculté, on utilise les notifications asynchrones, autrement dit l'interception d'évènements. Un autre mode de communication possible est l'utilisation du presse-papiers.

### *Communication directe*

Celle ci reprend tous les exemples que nous avons vus précédemment. Elle consiste en quelques sortes à écrire une macro dans le code Visual Basic. Nous regarderons plus loin, l'utilisation d'objets fournis par Excel pour faire de la programmation de VB. Voici un exemple classique de récupération de données Excel :

```
Private Sub TraiteExcel()  
Dim MonExcel As Excel.Application, MonClasseur As Excel.Workbook, MaFeuille  
As Excel.Worksheet  
Dim MonTab As Variant, MaPlage As Excel.Range  
Set MonExcel = New Excel.Application  
MonExcel.ReferenceStyle = xlR1C1  
Set MonClasseur =  
MonExcel.Workbooks.Open("D:\User\jmarc\tutorial\excel\tutor1.xls")  
Set MaFeuille = MonClasseur.Worksheets("pilotage")  
Set MaPlage = MaFeuille.UsedRange
```

```

ReDim MonTab(1 To MaPlage.Rows.Count, 1 To MaPlage.Columns.Count)
MonTab = MaPlage.Value
Set MaFeuille = Nothing
MonClasseur.Close False
Set MonClasseur = Nothing
MonExcel.Quit
Set MonExcel = Nothing
End Sub

```

Vous noterez que la variable "montab" est déclarée comme un variant et nom comme un tableau typé, sinon il y a le risque d'avoir une erreur lors de l'affectation. Notez que la ligne de redimensionnement est facultative.

### **Utiliser le presse-papiers**

Vous pouvez faire communiquer votre application avec Excel par l'intermédiaire du presse-papiers. Si je reprends le cas précédent avec :

```

MaPlage.Copy
MonTab = Clipboard.GetText

```

Je récupère une chaîne qui représente le tableau Excel. Ces chaînes sont toujours séparées avec des tabulations pour les colonnes et un retour chariot pour les lignes. La seule différence notable dans les deux cas est donc l'utilisation d'une chaîne ou d'un tableau.

### **Interception des événements**

Nous avons vu au début de cet article le codage de l'interception des événements. Sachez toutefois que pour que celui ci fonctionne correctement il faut utiliser des DoEvents dans votre code Visual Basic. Certains événements Post-opération (comme calculate) se prête particulièrement bien à la communication VB – Excel.

### **WorksheetFunction**

Un des avantages lors du pilotage d'Excel est de pouvoir accéder à l'objet WorksheetFunction dans le code Visual Basic. Certaines de ces fonctions acceptent comme arguments des variables n'étant pas de type Range et sont donc directement utilisables. Prenons l'exemple suivant :

```

Private Sub TraiteExcel()
Dim MonExcel As Excel.Application, TabNombre(1 To 100) As Integer
Dim compteur As Long, Moyenne As Single, Maxi As Integer, Mini As Integer
Set MonExcel = New Excel.Application
For compteur = 1 To 100
TabNombre(compteur) = compteur
Next compteur
Moyenne = MonExcel.WorksheetFunction.Average(TabNombre)
Maxi = MonExcel.WorksheetFunction.Max(TabNombre)
Mini = MonExcel.WorksheetFunction.Min(TabNombre)
MonExcel.Quit
Set MonExcel = Nothing
End Sub

```

J'utilise ainsi Excel comme une bibliothèque de fonctions supplémentaires pour Visual Basic..

### **Fonction renvoyant un tableau**

Certaines fonctions renvoient un tableau, comme la fonction Fréquence que nous allons voir. Celle-ci attend comme arguments un tableau d'origine et un tableau d'intervalles. Elle renvoie le tableau des distributions. On déclare le tableau de destination comme Variant.

```

Private Sub TraiteExcel()
Dim MonExcel As Excel.Application, MonTab As Variant, MesInter(2) As Integer
Dim TabNombre(1 To 100) As Integer, compteur As Long
Set MonExcel = New Excel.Application
For compteur = 1 To 100
If compteur Mod 2 = 0 Then TabNombre(compteur) = compteur Else
TabNombre(compteur) = 5
Next compteur
MesInter(0) = 10

```

```

MesInter(1) = 100
MonTab = MonExcel.WorksheetFunction.Frequency(TabNombre, MesInter)
Set MaFeuille = Nothing
MonClasseur.Close False
Set MonClasseur = Nothing
MonExcel.Quit
Set MonExcel = Nothing
End Sub

```

Attention, quelques fonctions attendent impérativement une plage, si on a besoin de les utiliser, il convient alors de transférer les données sur une feuille Excel. Comme ceci est assez lourd, essayez toujours de chercher une méthode de contournement.

### Correction orthographique

Nous allons voir un exemple simple d'utilisation dans VB du correcteur orthographique d'Excel.

```

Private Function TrouveFaute(PhraseTest As String) As Boolean
Dim MonExcel As Excel.Application
Set MonExcel = New Excel.Application
TrouveFaute = Not MonExcel.CheckSpelling(PhraseTest)
MonExcel.Quit
Set MonExcel = Nothing
End Function

```

## Astuces diverses

### Envoyer un classeur par mail

A condition d'avoir un système de messagerie installé par défaut, on peut envoyer directement un classeur par Mail. Le code VBA correspondant est :

```

Public Sub EnvoiClasseur()
If IsNull(Application.MailSession) Then Application.MailLogon "username",
"motdepasse"
ThisWorkbook.SendMail Recipients:="moi@toto.fr"
End Sub

```

### Bloquer les actions clavier & souris

On peut, pendant l'exécution d'un code assez long, bloquer le clavier et la souris afin que l'utilisateur ne puisse pas agir sur Excel pendant que celui ci travaille. Néanmoins, comme la méthode présente un risque il faut impérativement avoir un contrôle d'erreurs afin de pouvoir réactiver le clavier et la souris en cas de problème.

```

Public Sub LongTraitement()
On Error GoTo restauration
With Application
.DisplayAlerts = False
.ScreenUpdating = False
.Interactive = False
End With
'début du traitement long
restauration:
With Application
.DisplayAlerts = True
.ScreenUpdating = True
.Interactive = True
End With
End Sub

```

On utilise très souvent le mode Interactive False en pilotage VB. Le blocage du clavier n'empêche pas la saisie dans les UserForms.

## Ajouter du code à l'exécution

Pour finir cet article nous allons voir une méthode permettant d'ajouter un bouton à l'exécution et d'écrire l'événement correspondant pendant l'exécution. Cette méthode extrêmement puissante, puisqu'elle permet d'ajouter du code à la volée n'est pas sans risque, donc faite attention avant de l'utiliser.

Tout d'abord il faut ajouter au projet la référence MS Visual Basic for application "x.x" (Vbeext1.olb). Je vous met x.x car la version dépend de la version d'Excel.

Dans l'exemple qui suit, je vais ajouter un bouton de commande à la feuille et lui créer sa procédure d'événement click.

```
Public Sub AjoutBouton()  
Dim MaFeuille As Worksheet, MonBouton As Shape, PosLigne As Integer  
Set MaFeuille = ThisWorkbook.Worksheets("pilotage")  
Set MonBouton =  
MaFeuille.Shapes.AddOLEObject(ClassType:="Forms.CommandButton.1",  
Left:=100, Top:=100, Width:=100, Height:=200)  
MonBouton.Name = "CommandButton1"  
With ThisWorkbook.VBProject.VBComponents("Feuil4").CodeModule  
.CreateEventProc "Click", "CommandButton1"  
PosLigne = .ProcStartLine("CommandButton1_Click", vbext_pk_Proc)  
.InsertLines PosLigne + 3, "msgbox " & Chr(34) & "nouveau bouton" &  
Chr(34)  
End With  
End Sub
```

## Conclusion

Comme nous l'avons vu la programmation d'Excel n'est pas très compliquée, encore faut il être rigoureux.

Gardez toujours à l'esprit qu'il faut toujours privilégier la vitesse d'exécution celle-ci étant le problème majeur du VBA Excel. Dans de nombreux cas, il y a plusieurs moyens pour arriver au même résultat, n'hésitez pas à tester les diverses méthodes que vous imaginez.

Bonne programmation.