Formation Excel - VBA débutant

Introduction

L'ensemble des logiciels de la suite Microsoft® Office utilise un langage de programmation intégré appelé Visual Basic for Applications (VBA). Il s'agit d'un langage Visual Basic simplifié couplé au modèle objet de l'application office qui le contient.

Nous allons dans ce cours voir ou revoir les bases de la programmation Visual basic et la manipulation du modèle objet de Microsoft® Excel.

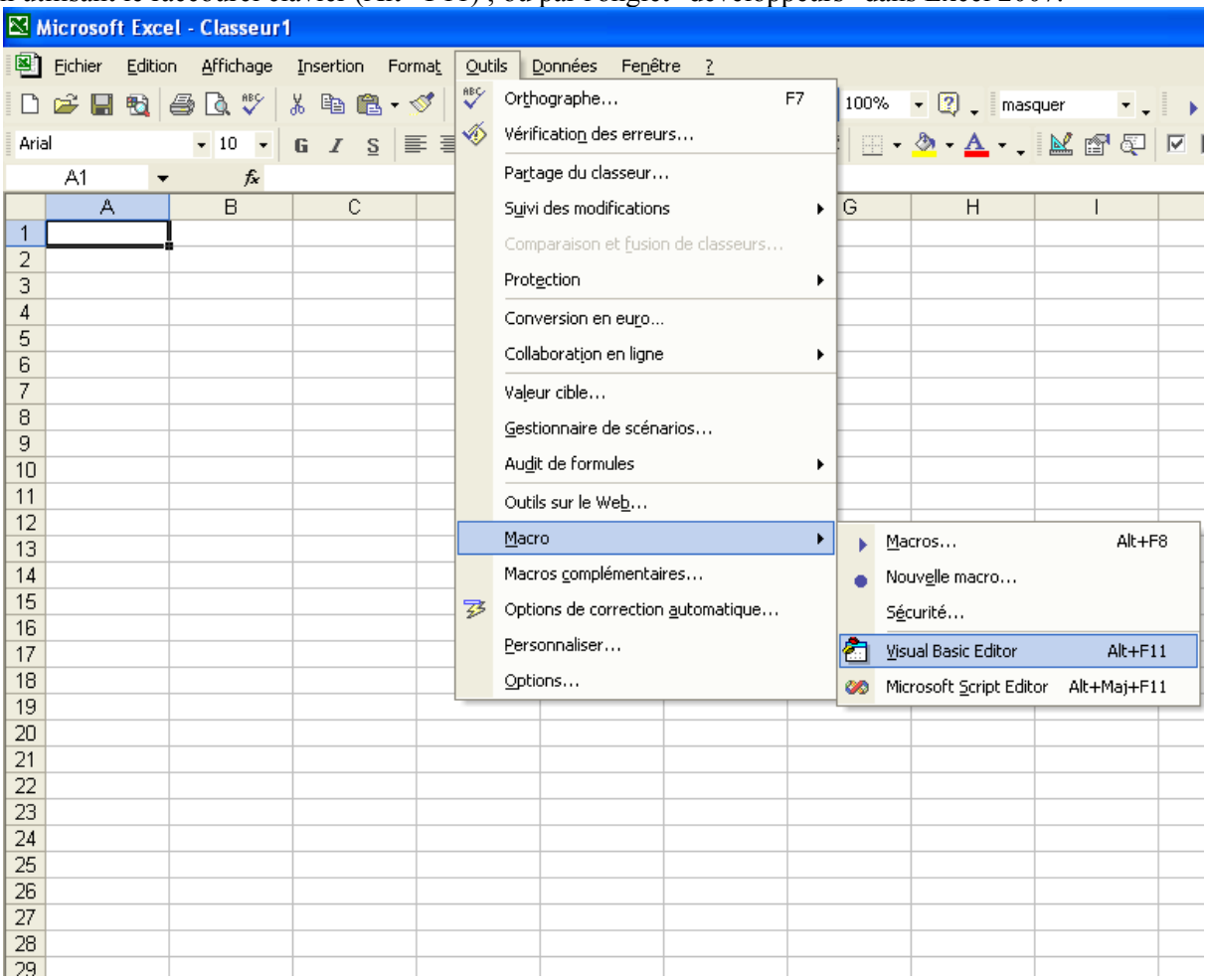
Pour les nouveaux développeurs

Si vous n'avais jamais approché de près ou de loin un langage informatique, vous risquez de trouver le début de ce cours extrêmement complexe. Certains concepts évoqués au début de ce cours ne seront abordés que plus loin dans celui-ci. Lisez le une fois rapidement sans entrer dans le détail, cela devrez vous permettre de vous imprégner de la terminologie.

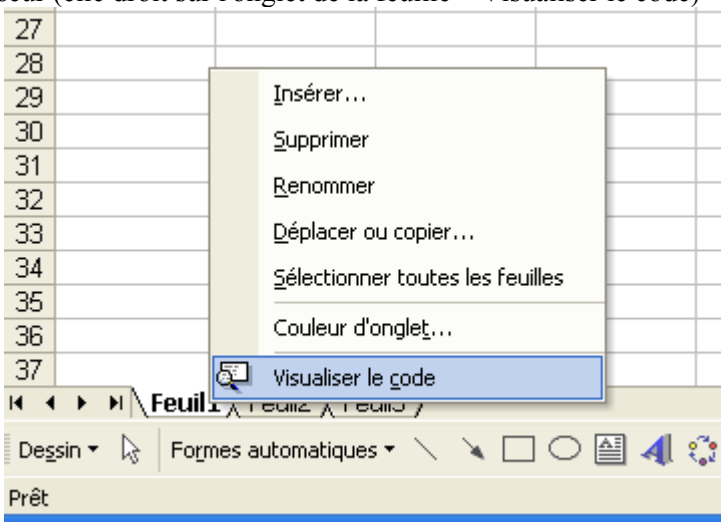
L'environnement de développement

L'environnement de développement de VBA est intégré à l'application Office. Il existe deux façons d'y accéder volontairement et une bonne quinzaine d'y accéder sans le vouloir. Les deux façons sont un peu différentes puisqu'on n'arrive pas dans le même module selon les cas.

Généralement on accède à l'éditeur en choisissant le menu "Outils – Macro – Visual Basic Editor" ou en utilisant le raccourci clavier (Alt +F11) ; ou par l'onglet "développeurs" dans Excel 2007.



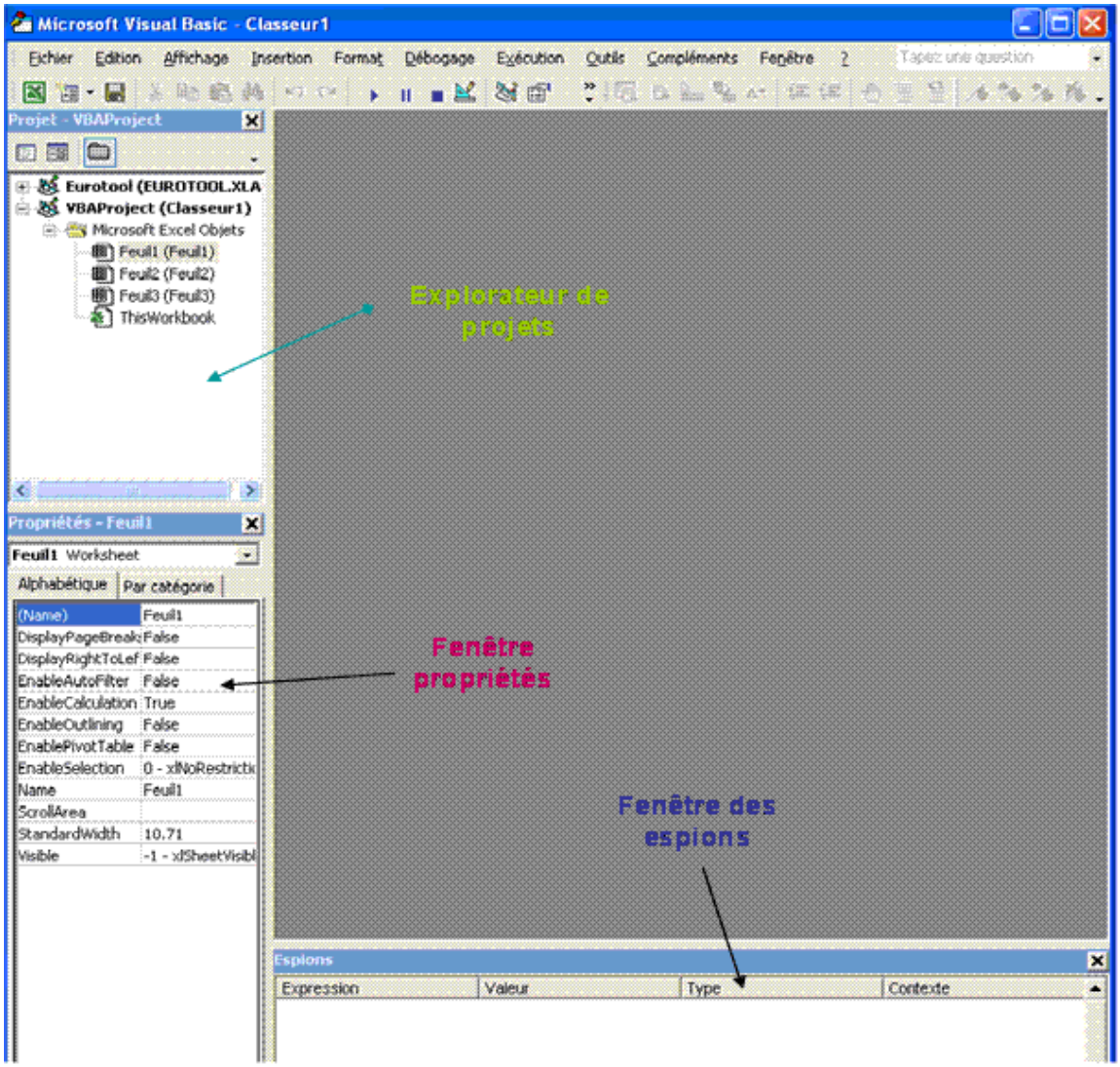
Cependant on peut aussi y accéder en utilisant le menu contextuel des onglets de feuilles du classeur (clic droit sur l'onglet de la feuille – Visualiser le code)



Dans ce cas nous arriverons dans le module de code de la feuille sélectionnée.

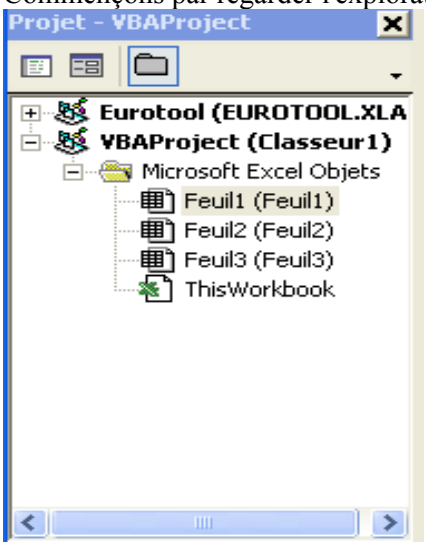
Présentation de l'éditeur

L'éditeur ressemble à :



Notions de module

Commençons par regarder l'explorateur de projet.

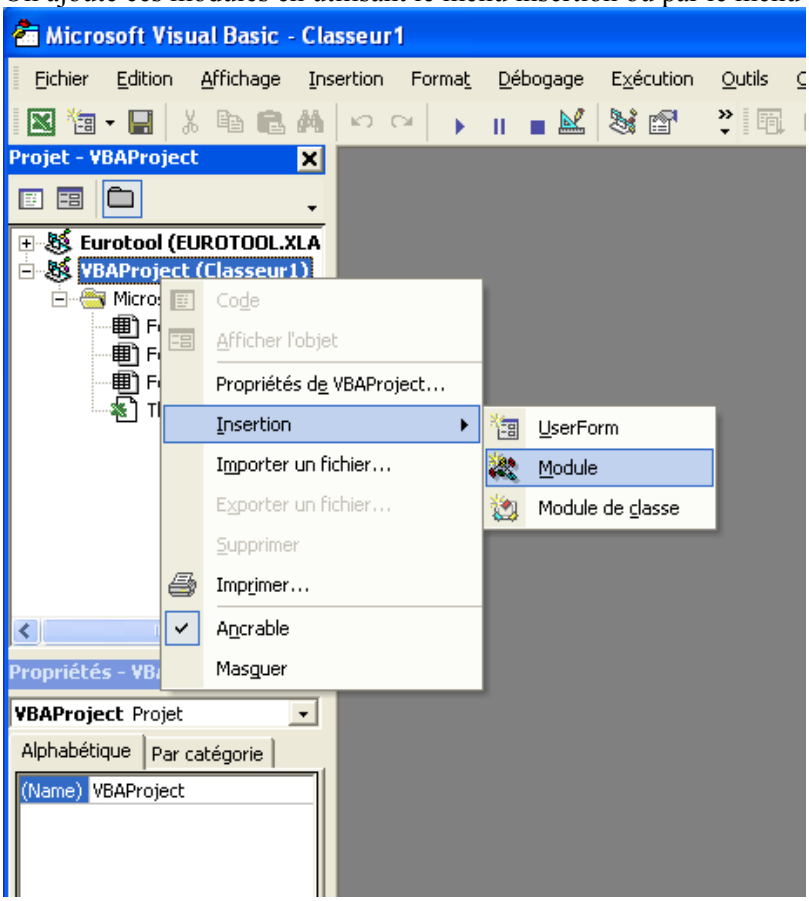


L'explorateur va afficher l'ensemble des projets en cours. Chaque classeur contient un projet. Un projet contient des modules de codes, c'est-à-dire des unités logiques pouvant contenir du code. Par défaut comme vous le voyez sur la figure ci-dessus, il y a un module de code par feuilles contenues dans le classeur et un module pour le classeur intitulé "ThisWorkbook".

Il est possible d'ajouter aux projets des modules supplémentaires qui peuvent être de trois types :

- Les modules standards
- Les formulaires (UserForms)
- Les modules de classe

On ajoute ces modules en utilisant le menu insertion ou par le menu contextuel du projet.

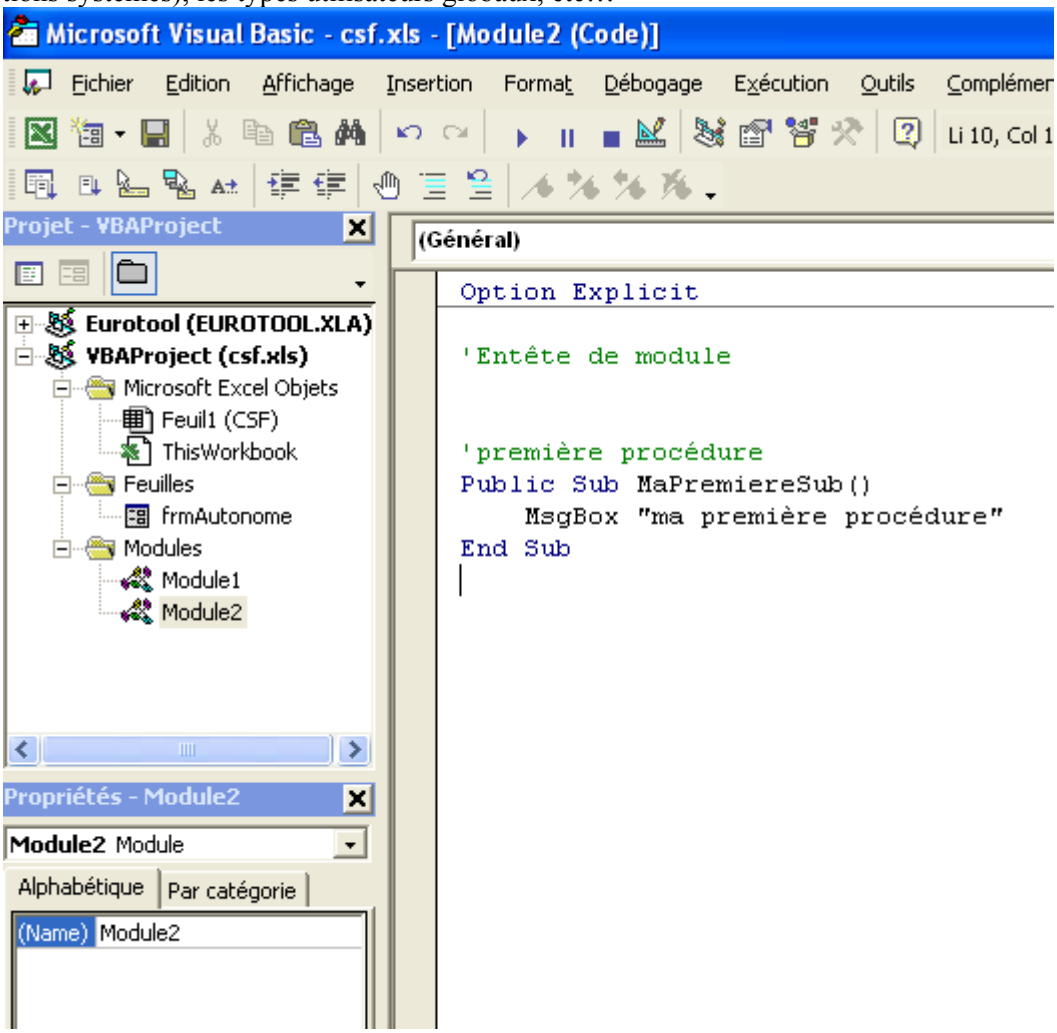


A l'exception des modules standards, tous les autres modules sont dit modules objets c'est-à-dire acceptant la déclaration de variables objets sensibles aux évènements; c'est-à-dire qui gèrent du code évènementiel). Cette notion sera vue succinctement en fin de cours.

Le code Visual Basic est toujours contenu dans un module. Il peut être contenu dans plusieurs modules du même projet en suivant les règles suivantes :

1. Le code générique, le code mettant en cause plusieurs objets du classeur, les fonctions accessibles dans tout le projet doivent être dans un module standard.
2. Les UserForms ne contiennent que le code de leur propre fonctionnement et éventuellement la fonction d'appel.
3. Chaque classe est dans un module de classe distinct
4. Le module ThisWorkbook ne contient que le code évènementiel du classeur et les fonctions privées éventuelles
5. Chaque module de feuille ne contient que le code évènementiel de la feuille et les fonctions privées éventuelles ainsi que le code évènementiel des objets sensibles aux évènements qu'elle contient.
6. On peut utiliser plusieurs modules standards pour regrouper les fonctionnalités connexes

La partie haute du module située entre la déclaration des options et la définition de la première procédure est appelée entête de module (parfois tête de module). C'est dans cette partie que sont déclarées les variables globales, les déclarations d'API (Application Programming Interface ou fonctions systèmes), les types utilisateurs globaux, etc…



L'enregistreur de macro

L'enregistreur de macro est un utilitaire d'écriture de code contenu dans l'application office. On l'utilise en sélectionnant "Outils – Macro – Nouvelle Macro", ce qui déclenche l'affichage de la boîte suivante :

Le code généré suit strictement les actions de l'utilisateur. Ainsi, le code suivant va écrire des valeurs de 1 à 10 dans la plage A1:A10 :

```
Sub Macro1()
'
' Macro1 Macro
' Macro enregistrée le 20/06/2002 par XXXX
'

'
    Range("A1").Select
    ActiveCell.FormulaR1C1 = "1"
    Range("A2").Select
    ActiveCell.FormulaR1C1 = "2"
    Range("A1:A2").Select
    Selection.AutoFill Destination:=Range("A1:A10"), Type:=xlFillDefault
    Range("A1:A10").Select
End Sub
```

Si le côté pratique de l'enregistreur est difficilement contestable, le code généré est de très mauvaise qualité en terme d'efficacité et de lisibilité. Par exemple l'écriture correcte du code ci-dessus serait :

```
Sub Macro1()

    With Cells(1, 1)
        .Value = 1
        .Resize(10).DataSeries Rowcol:=xlColumns, Type:=xlLinear, Step:=1,
Stop:=10
    End With

End Sub
```

Cependant l'enregistreur de macro est utile pour retrouver une syntaxe ou pour examiner les méthodes utilisées lors d'un enchaînement d'action.

Visual Basic

Présentation

Visual Basic est un environnement de développement intégré propriétaire pour le langage BASIC sous Windows, édité par Microsoft™. Il en existe plusieurs groupes qui sont

Jusqu'à la version 4 et pour toutes les versions VBA – Interprétés / évènementiels procéduraux

Les versions 5 et 6 – compilés (ou natifs) / évènementiels procéduraux

A partir de VB 2003 (VB.NET) – managés / objets

Pour faire simple, les langages interprétés ont besoin d'un 'runtime' spécifique pour s'exécuter, l'interpréteur de commande. A l'identique des langages de script, le code est interprété sous la forme dans lequel vous l'écrivez, bien qu'il soit possible de lui faire subir une pseudo compilation pour rechercher certains types d'erreurs (de liaison principalement).

Un langage natif (ou compilé) transforme le code que vous écrivez à l'aide d'un compilateur en une série d'instruction directement utilisable par le processeur.

Un langage managé utilise un hôte d'exécution pour sécuriser son fonctionnement et le système d'exploitation.

Les langages procéduraux utilisent des éléments de codes sous forme de procédures linéaires.

Les langages évènementiels utilisent des éléments de code répondant à des évènements spécifiques.

Les langages objets utilisent des éléments de codes décrits dans des classes.

Les variables

En développement, on entend par variable une donnée définie dans un contexte donné ayant un type défini. Autrement dit, une variable est la représentation d'une valeur au sens large du terme.

On appelle déclaration le fait de définir la variable avant de l'utiliser, dimensionnement : le fait de lui donner un type.

En Visual Basic, la déclaration des variables n'est pas obligatoire tant que l'option Explicit n'est pas activée. Le dimensionnement n'est jamais obligatoire puisque les variables ont toujours à minima le type universel par défaut.

La déclaration des variables en Visual Basic est de la forme suivante :

Instruction de déclaration – Nom de la variable – As – Type de la variable

Par exemple :

```
Dim Age As Integer
```

Déclare la variable Age comme étant un entier 16 bits.

On peut utiliser une seule instruction de déclaration pour déclarer plusieurs variables en les séparant par des virgules. Par exemple la ligne suivante déclare et dimensionne deux variables Nom et Prenom comme étant deux chaînes de caractères :

```
Dim Nom As String, Prenom As String
```

Contrairement à de nombreux autres langages, le type ne se propage pas sur une ligne de déclaration. Ainsi la déclaration :

```
Dim Nom, Prenom As String
```

N'est pas équivalente à la déclaration précédente puisqu'elle se lit :

```
Dim Nom As Variant, Prenom As String
```

Comme dit précédemment, la déclaration et le dimensionnement sont facultatifs par défaut. Les codes suivants sont donc tous valides :

Sans déclaration

```
Sub CalculPerimetre()
   Rayon = InputBox("Entrez le rayon en mm", "RAYON", 0)
   circonference = 2 * 3.14159 * Rayon
   MsgBox "le périmètre est de " & circonference & " mm"

End Sub
```

Sans dimensionnement

```
Sub CalculPerimetre()

Dim Rayon, Circonference
   Rayon = InputBox("Entrez le rayon en mm", "RAYON", 0)
   Circonference = 2 * 3.14159 * Rayon
   MsgBox "le périmètre est de " & Circonference & " mm"

End Sub
```

Typés

```
Sub CalculPerimetre()

Dim Rayon As Integer, Circonference As Single
Rayon = InputBox("Entrez le rayon en mm", "RAYON", 0)
Circonference = 2 * 3.14159 * Rayon
MsgBox "le périmètre est de " & Circonference & " mm"

End Sub
```

S'ils sont tous valides, ils n'en sont pas équivalents pour autant. Si vous exécutiez le premier et le dernier code en saisissant 10.2 comme valeur de rayon, le premier code renverrait une valeur et le dernier une erreur.

Les noms de variable doivent commencer par un caractère alphabétique, être uniques au sein d'une même portée, ne doivent pas excéder 255 caractères et ne peuvent contenir ni caractère de déclaration de type ni point.

La portée

La notion de portée, parfois appelée visibilité, défini les limites d'accessibilité d'une variable. Il existe plusieurs instructions de déclaration selon la portée désirée et la déclaration ne se fait pas au même endroit.

Instruction	Déclaration	Commentaires
Private	Module	Visible par tout le code du module mais inaccessible depuis un autre module
Public	Module (standard)	Visible par tout le code du projet. Ne se déclare que dans les modules standard.
Dim	Fonction	Uniquement dans la fonction ou elle est déclarée. Si utilisée au niveau module, équivaut à Private
Static	Fonction	Uniquement dans la fonction ou elle est déclarée. N'est pas détruite à la fin de la fonction

Imaginons le cas suivant. Dans un module standard je déclare :

```
Public VarPublicModuleStd As String
Private VarPrivateModuleStd As String
```

Dans le module " Feuil1 ", j'ajoute un bouton sur la feuille et dans le module de code, j'écris :

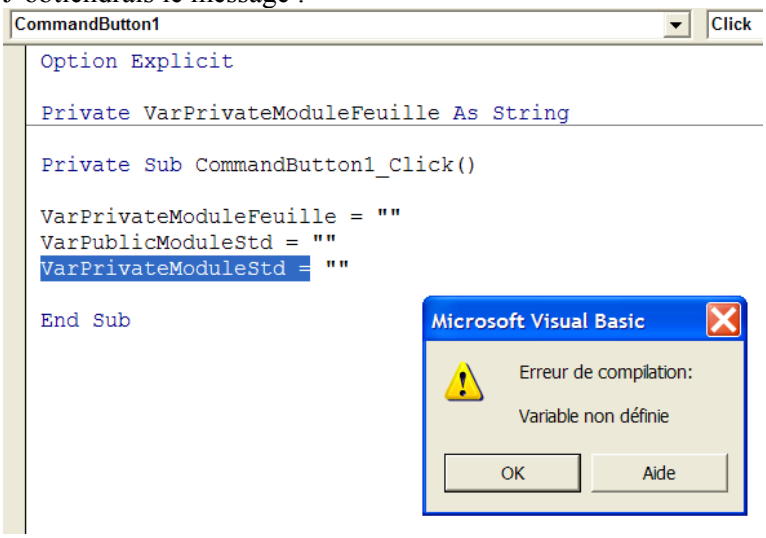
```
Private VarPrivateModuleFeuille As String

Private Sub CommandButton1_Click()

VarPrivateModuleFeuille = ""
VarPublicModuleStd = ""
VarPrivateModuleStd = ""

End Sub
```

Si nous avons bien compris les règles de la portée, nous allons obtenir une erreur sur la troisième ligne puisque la variable est privée dans le module standard. Pourtant le code s'exécute sans erreur.

Comme nous l'avons vu, sauf stipulation contraire, Visual Basic ne force pas la déclaration des variables. Dans ce cas, comme la fonction ne voit pas de déclaration pour VarPrivateModuleStd, elle en crée une implicite ce qui fait que le code s'exécute mais qu'on ne travaille pas avec la variable VarPrivateModuleStd du module standard. Par contre si j'écris :

```
Option Explicit

Private VarPrivateModuleFeuille As String

Private Sub CommandButton1_Click()
   VarPrivateModuleFeuille = ""
   VarPublicModuleStd = ""
   VarPrivateModuleStd = ""
End Sub
```

J'obtiendrais le message :



Et la variable incriminée sera surlignée.

Cette portée induit la notion de durée de vie des variables. Les variables de niveaux modules sont dites permanentes dans le sens où elles existent tant que le code s'exécute. On les appelle aussi variables globales. Les variables de niveau fonction n'existent que lorsque la fonction s'exécute et sont détruites quand la fonction se termine. Elles ne sont donc jamais accessibles en dehors de la fonction où elles sont déclarées. On les appelle variables locales. Normalement elles sont détruites en fin de fonction et perdent leur valeur, mais les variables statiques (déclarées avec l'instruction Static) la conserve. Un appel ultérieur à la fonction permettra de retrouver la variable dans l'état où elle était à la fin de l'appel précédent.

Regardons l'exemple suivant :

```
Private Sub CommandButton2_Click()
Dim VarLocale As Integer
   VarLocale = VarLocale + 1
   MsgBox VarLocale
End Sub

Private Sub CommandButton3_Click()
Static VarLocale As Integer
   VarLocale = VarLocale + 1
   MsgBox VarLocale
End Sub
```

Plusieurs clics sur le bouton 2 provoqueront toujours l'affichage de la valeur "1" alors que plusieurs clics sur le bouton 3 provoqueront l'affichage d'une valeur incrémentée de 1 à chaque clic. Notez que la variable locale des deux fonctions peut avoir le même nom puisqu'elles sont hors de portée l'une de l'autre.

Le type

Le type d'une variable c'est la détermination du genre de valeur que la variable peut contenir. En VBA, toutes les variables possèdent le type *Variant* par défaut, appelé parfois type universel. Une variable de type Variant peut contenir n'importe qu'elle valeur à l'exception des chaînes de longueur fixe. Les variables de type Variant peuvent aussi avoir des valeurs particulières, à savoir Empty, Error et Nothing. Nous verrons les significations de ces valeurs plus loin dans ce cours.

Les types de données utilisés en VBA sont :

Nom		Taille (o)	Conversion
Byte	0 à 255	1	CByte
Boolean	True (<>0) False(0)	2	CBool
Integer	-32 768 à 32 767	2	CInt
Long	-2 147 483 648 à -2 147 483 647	4	CLng
Single	-3,402823E38 à -1,401298E-45 pour les valeurs négatives ; 1,401298E-45 à 3,402823E38 pour les valeurs positives	4	CSng
Double	-1,79769313486231E308 à -4,94065645841247E-324 pour les valeurs négatives ; 4,94065645841247E-324 à 1,79769313486232E308 pour les valeurs positives	8	CDbl
Currency	-922 337 203 685 477,5808 à 922 337 203 685 477,5807	8	CCur
Decimal	+/-79 228 162 514 264 337 593 543 950 335 sans séparateur décimal +/-7,9228162514264337593543950335 avec 28 chiffres à droite du séparateur décimal le plus petit nombre différent de zéro est +/-0.0000000000000000000000000001	14	CDec
Date	1er janvier 100 au 31 décembre 9999	8	CDate
Object	Tous les objets	4	
String (fixe)	65536 caractères	Nombre caractères	
String (var)	2 147 483 648 caractères (2^31)	10 + Nombre caractères	Cstr
Variant (nombre)	Même plage que Double	16	Cvar
Variant (chaîne)	Même plage que chaîne variable	22 + Nombre caractères	CVar

Conversion de type

La conversion de type est l'opération qui consiste à convertir une expression en un type de donnée défini. En développement, on entend par expression une combinaison de mots clés, d'opérateurs, de variables et de constantes générant une chaîne, un nombre ou un objet. Une expression peut effectuer un calcul, manipuler des caractères ou tester des données.

Les règles suivantes s'appliquent :

- Si l'argument *expression* passé à la fonction excède la plage de valeurs du type de données cible, une erreur se produit.
- Il est généralement possible de documenter le code en utilisant les fonctions de conversion de types de données afin d'indiquer que le résultat de certaines opérations devrait correspondre à un type de données particulier plutôt qu'au type par défaut. Utilisez par exemple la fonction **CCur** pour fonctionner en arithmétique monétaire et non en arithmétique en simple précision, en double précision ou en arithmétique de nombres entiers.
- Utilisez les fonctions de conversion de types de données à la place de la fonction **Val** de manière à respecter les conventions étrangères. Par exemple, la fonction **CCur** reconnaît divers types de séparateurs décimaux, de séparateurs des milliers et diverses options monétaires, selon les paramètres régionaux de votre ordinateur.
- Les fonctions **CInt** et **CLng** arrondissent les parties décimales égales à 0,5 au nombre pair le plus proche. Par exemple, 0,5 est arrondi à 0 et 1,5 est arrondi à 2. Les fonctions **CInt** et **CLng** diffèrent des fonctions **Fix** et **Int**, qui tronquent la partie décimale d'un nombre sans forcément l'arrondir. En outre, les fonctions **Fix** et **Int** renvoient toujours une valeur du type passé en argument.
- Utilisez la fonction **IsDate** pour déterminer si la valeur de l'argument *date* peut être convertie en date ou en heure. La fonction **CDate** reconnaît les littéraux date et heure ainsi que certains nombres appartenant à la plage de dates autorisées. Lors de la conversion d'un nombre en date, la partie entière du nombre est convertie en date. Si le nombre comprend une partie décimale, celle-ci est convertie en heures, exprimées en partant de minuit.
- La fonction **CDate** reconnaît les formats de date définis dans les paramètres régionaux de votre système. L'ordre des jours, mois et années risque de ne pouvoir être défini si les données sont fournies dans un format différent des paramètres de date reconnus. En outre, les formats de date complets précisant le jour de la semaine ne sont pas reconnus.

Dans de nombreux cas, VBA va exécuter de lui-même des conversions de type dites implicites. Ces conversions doivent toujours être évitées soit en explicitant la conversion, soit en typant correctement les variables.

Les constantes

Tel que leur nom l'indique, les constantes sont des variables qui ne varient pas. Elles se déclarent à l'aide de l'instruction *Const*, peuvent utiliser une instruction de portée et un type. Par exemple dans notre fonction précédente, nous pourrions écrire :

```
Public Const PI As Single = 3.14159

Sub CalculPerimetre()

Dim Rayon As Integer, Circonference As Single
Rayon = InputBox("Entrez le rayon en mm", "RAYON", 0)
Circonference = 2 * PI * Rayon
MsgBox "le périmètre est de " & Circonference & " mm"

End Sub
```

Intérêts

Où est donc l'intérêt de dimensionner et a fortiori de déclarer ?

Pour le dimensionnement, il s'agit principalement d'économiser de la mémoire et du temps d'exécution. Cependant il s'agit aussi d'une certaine garantie contre un risque d'erreur de logique. Enfin la lisibilité du code n'en est que meilleure. Par ailleurs, le type Variant utilise des conversions par défaut qui peuvent produire un résultat inattendu avec certains opérateurs. Imaginons le code suivant :

```
Sub ConversionArbitraire()

Dim VarNonTypee As Variant
    VarNonTypee = 3
    MsgBox VarNonTypee + 3
    VarNonTypee = "coucou"
    MsgBox VarNonTypee + 3

End Sub
```

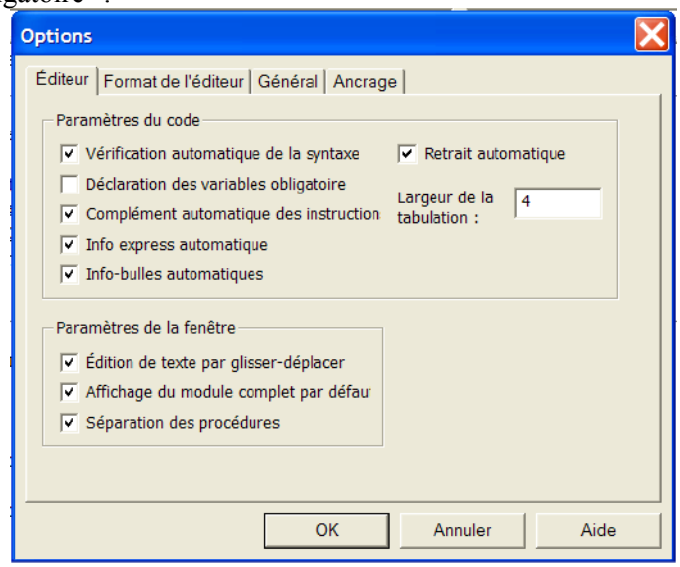
Si vous exécutez ce code, vous allez obtenir comme résultat 6 puis coucou3. Notez que ces conversions ne sont normalement plus possibles dans les dernières versions d'Excel.

L'intérêt de la déclaration, tout au moins de la déclaration forcée est beaucoup plus facile à démontrer. Reprenons notre exemple :

```
Sub CalculPerimetre()
   Rayon = InputBox("Entrez le rayon en mm", "RAYON", 0)
   Ciconference = 2 * PI * Rayon
   MsgBox "le périmètre est de " & Circonference & " mm"
End Sub
```

Quelle que soit la valeur rentrée dans la boîte de saisie, le résultat sera toujours 0. Comme vous l'avez peut être remarqué, il y a une faute de frappe dans le nom de la variable Circonférence, et comme VBA ne force pas la déclaration, il crée une deuxième variable qui elle contient 0.

Notez que la déclaration des variables ne résoudrait pas le problème, sauf si vous êtes un as du déboguage. En effet, il faut préciser à VBA qu'on souhaite travailler en déclaration forcée pour que celui-ci contrôle la déclaration des variables. Pour obtenir cela, on tape Option Explicit en haut du module, ou on va dans le menu Outils – Option et on coche la case " déclaration des variables obligatoire ".



Dès lors, notre fonction avec une faute de frappe ne fonctionnera plus puisque la variable n'est pas déclarée.

Notons aussi que la déclaration obligatoire des variables augmente grandement la lisibilité du code.

Le type Variant

Appelé parfois improprement "type universel", le type Variant est un type union c'est-à-dire pouvant avoir plusieurs représentations d'une même variable ou acceptant plusieurs types de variables. En Visual Basic, le type variant peut contenir tous types de variable. Quoiqu'il soit parfois très utile, il convient de ne pas abuser de son utilisation. En effet, il demande plus de ressources que les autres types ce qui ralentit l'exécution des programmes. Par ailleurs la plupart des erreurs induites par l'utilisation de ce type se produisent à l'exécution ce qui tend à augmenter la quantité de code de gestion d'erreurs et à complexifier le déboguage.

Cependant, il est inévitable de savoir correctement l'appréhender en VBA puisque par définition, les valeurs des cellules Excel sont de types Variant.

Le type Variant peut contenir tout type de valeurs, notamment :

- Un nombre
- Une chaîne de caractères
- Un booléen
- Un tableau
- Un objet
- Une valeur particulière
  - Empty : la variable est vide
  - Nothing : Objet non initialisé
  - NULL : Valeur vide d'une base de données
  - Error : Une valeur d'erreur

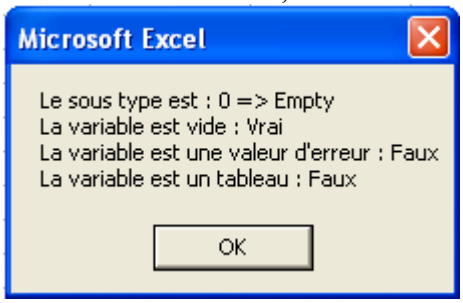
Visual Basic met en place un certain nombre de fonctions spécifiques pour travailler sur les variants :

- IsArray renvoie vrai si la variable est un tableau
- IsEmpty renvoie vrai si la variable est vide
- IsError renvoie vrai si la variable est une valeur d'erreur
- VarType renvoie un entier identifiant le sous type de la variable
- TypeName renvoie une chaîne identifiant le sous type de la variable.

On pourrait imaginer un code de test tel que :

```
Sub TestVariant()

Dim MaVar As Variant, Message As String

MaVar = Range("B2").Value
Message = "Le sous type est : " & VarType(MaVar) & " => " & TypeName(MaVar)
& vbNewLine
Message = Message & "La variable est vide : " & IsEmpty(MaVar) & vbNewLine
Message = Message & "La variable est une valeur d'erreur : " &
IsError(MaVar) & vbNewLine
Message = Message & "La variable est un tableau : " & IsArray(MaVar)
MsgBox Message

End Sub
```

Si la cellule B2 est vide, ce code affichera la boîte de message suivante :



Type utilisateur

Il est possible de définir des types composites, appelés types utilisateurs à l'aide de l'instruction Type… End Type.

L'intérêt est évidemment de manipuler plusieurs variables connexes à l'aide d'une seule variable. Ce type étant ensuite considéré comme n'importe quel type, vous pouvez déclarez des variables de ce type, des tableaux, le renvoyer dans des fonctions, etc…

La définition d'un type utilisateur se fait obligatoirement au niveau du module. Dans un module standard, il peut être public ou privé, dans un module objet il ne peut être que privé.

La déclaration se fait sous la forme :

Portee Type *NomType*
Element As Type
Element As Type
….
End Type

Les éléments qui composent le type (appelés membres) peuvent être de n'importe quels types prédéfinis, des tableaux ou d'autres types utilisateurs. Il est donc possible d'obtenir des structures extrêmement complexes.

N'oubliez pas que cette définition ne suffit pas pour manipuler la structure, vous devez déclarer des variables de ce type pour l'utiliser effectivement.

Pour accéder aux membres, c'est l'opérateur "." (Point) qui est utilisé.

```
Public Type Fichier
    Nom As String
    Repertoire As String
    DateCration As Date
    Taille As Long
End Type

Public Sub test()

Dim FichiersExcel() As Fichier, compteur As Long, fTemp As String

fTemp = Dir("d:\svg\jmarc\*.xls", vbNormal)
Do Until fTemp = ""
    ReDim Preserve FichiersExcel(0 To compteur)
    FichiersExcel(compteur).Nom = fTemp
    FichiersExcel(compteur).Repertoire = "d:\svg\jmarc\*.xls"
    FichiersExcel(compteur).DateCration = FileDateTime("d:\svg\jmarc\" &
fTemp)
    FichiersExcel(compteur).Taille = FileLen("d:\svg\jmarc\" & fTemp)
    compteur = compteur + 1
    fTemp = Dir
Loop
MsgBox FichiersExcel(0).Nom & vbNewLine & FichiersExcel(0).Taille

End Sub
```

Énumération

Une énumération est un groupement de constantes entières connexes. Elle est toujours déclarée au niveau du module. L'intérêt repose surtout sur la lisibilité du code, on peut indifféremment utiliser la valeur numérique ou le membre de l'énumération. Elle se déclare comme suit :

Portée Enum *Name*
NomMembre = [*ConstanteEntiere*]
NomMembre = [*ConstanteEntiere*]
….
End Enum

Les valeurs d'une énumération peuvent être utilisées soit directement, soit à l'aide de variables, passées comme argument ou renvoyées par une fonction. Si les valeurs des constantes ne sont pas précisées dans la déclaration, le premier membre vaut zéro, chaque membre suivant étant égal à la valeur du membre précédent agrémenté de 1.

VBA utilise de nombreuses énumérations intégrées.

```
Public Enum Mois
    Janvier = 1
    Février = 2
    Mars = 3
    Avril = 4
    Mai = 5
    Juin = 6
    Juillet = 7
    Aout = 8
    Septembre = 9
    Octobre = 10
    Novembre = 11
    Décembre = 12
End Enum

Public Enum Facteur
    kilo = 1024
    mega = 1048576
    giga = 1073741824
End Enum

Private Function LastDayOfMonth(ByVal Annee As Integer, ByVal LeMois As
Mois) As Integer

    LastDayOfMonth = Day(DateSerial(Annee, LeMois + 1, 0))

End Function
Public Sub TestEnum()
Dim Taille As Long

    Taille = FileLen("d:\svg\jmarc\setup.exe")
    Call MsgBox("Taille : " & Taille & " (" & Taille / Facteur.mega & "
Mo)")
    Call MsgBox("dernier jour de février 2000 : " & LastDayOfMonth(2000,
Février))

End Sub
```

Masque binaire

Les masques binaires sont une forme particulière d'énumération. Il repose sur la représentation binaire des nombres entiers. En effet, on peut représenter un nombre entier sous la forme d'une succession de valeur zéro ou un, qu'on peut assimiler à une suite de champs booléens. Il est alors possible de stocker un grand nombre d'informations dans une structure relativement compacte.

Un masque binaire sera donc une énumération dont chaque membre aura comme valeur une puissance de deux afin que la composition des membres donne forcément une valeur unique.

Nous reviendrons plus en détail sur leur fonctionnement un peu plus loin dans ce cours

Opérateurs

Aussi incroyable que cela puisse paraître, les opérateurs servent à réaliser des opérations. Il existe relativement peu d'opérateurs en VBA, qui ont divisés en quatre grandes familles.

Opérateurs arithmétiques

Opérateur	Commentaire
+	Addition de deux nombres
-	Soustraction de deux nombres
*	Multiplication de deux nombres
/	Division de deux nombres, le dénominateur ne peut être nul.
\	Division entière. Renvoie la partie entière du résultat
^	Élévation à la puissance du nombre de gauche par celui de droite
Mod	Renvoie le reste de la division du nombre de gauche par celui de droite.

Donc en prenant le code suivant :

```
Public Sub OperateurArithmetique()

Dim Operande1 As Integer, Operande2 As Integer, Result As Double

    Operande1 = 5
    Operande2 = 2
    Result = Operande1 + Operande2
    Debug.Print Result
    Result = Operande1 - Operande2
    Debug.Print Result
    Result = Operande1 * Operande2
    Debug.Print Result
    Result = Operande1 / Operande2
    Debug.Print Result
    Result = Operande1 \ Operande2
    Debug.Print Result
    Result = Operande1 ^ Operande2
    Debug.Print Result
    Result = Operande1 Mod Operande2
    Debug.Print Result

End Sub
```

Nous obtiendrions dans la fenêtre d'exécution :

```
7
3
10
2.5
2
25
1
```

Opérateurs de comparaison

Les opérateurs de comparaison sont tout aussi connus que les opérateurs arithmétiques

Opérateur	vrai si	faux si
< (inférieur à)	*expression1* < *expression2*	*expression1* >= *expression2*
<= (inférieur ou égal à)	*expression1* <= *expression2*	*expression1* > *expression2*
> (supérieur à)	*expression1* > *expression2*	*expression1* <= *expression2*

Opérateur	vrai si	faux si
>= (supérieur ou égal à)	*expression1* >= *expression2*	*expression1* < *expression2*
= (égal à)	*expression1* = *expression2*	*expression1* <> *expression2*
<> (différent de)	*expression1* <> *expression2*	*expression1* = *expression2*

Il est toujours important de savoir cependant ce que l'on compare car VBA compare différemment les chaînes des nombres. Par exemple la chaîne "10" et inférieure à la chaîne "2" alors que c'est l'inverse pour les valeurs numériques.

Les règles suivantes s'appliquent :

Condition	Résultat
Les deux expressions sont des types de données numériques (Byte, Boolean, Integer, Long, Single, Double, Date, Currency ou Decimal)	Effectue une comparaison numérique.
Les deux expressions sont de type String	Effectue une comparaison de chaînes.
Une expression est de type numérique et l'autre de type **Variant**, qui est ou peut-être un nombre	Effectue une comparaison numérique.
Une expression est de type numérique et l'autre est une chaîne de type **Variant** ne pouvant être convertie en nombre	L'erreur `type incompatible` se produit.
Une expression est de type **String** et l'autre de type **Variant**, sauf **Null**	Effectue une comparaison de chaînes.
Une expression est **Empty** et l'autre est de type numérique	Effectue une comparaison numérique en utilisant la valeur 0 pour l'expression **Empty**.
Une expression est **Empty** et l'autre est de type **String**	Effectue une comparaison de chaînes en utilisant une chaîne de longueur nulle ("") pour l'expression **Empty**.

Si les arguments *expression1* et *expression2* sont tous deux de type **Variant**, la méthode de comparaison est déterminée par leur type sous-jacent. Le tableau suivant définit la méthode de comparaison des expressions de type **Variant** et les résultats obtenus en fonction du type sous-jacent :

Condition	Résultat
Les deux expressions de type **Variant** sont numériques	Effectue une comparaison numérique.
Les deux expressions de type **Variant** sont des chaînes	Effectue une comparaison de chaînes.
Une des expressions de type **Variant** est numérique et l'autre est une chaîne	L'expression numérique est inférieure à l'expression de chaîne.
Une des expressions de type **Variant** a la valeur **Empty** et l'autre est numérique	Effectue une comparaison numérique en utilisant 0 pour l'expression **Empty**.
Une des expressions de type **Variant** est **Empty** et l'autre est une chaîne	Effectue une comparaison numérique en utilisant une chaîne de longueur nulle ("") pour l'expression **Empty**.
Les deux expressions de type **Variant** sont **Empty**	Les expressions sont égales.

Il existe deux types de comparaison de chaînes, la comparaison binaire qui tient compte de la casse et la comparaison texte qui n'en tient pas compte. On peut définir la comparaison par défaut à la valeur texte à l'aide d'"Option Compare Text" en tête de module.

Deux opérateurs de comparaison spécifique existent aussi. L'opérateur Is sera vu dans la partie de ce cours traitant des objets. L'opérateur Like permet une comparaison de chaîne avec un modèle de chaîne acceptant des caractères génériques (*, ?).

&, Opérateur de concaténation

L'opérateur & permet de concaténer (réunir) deux chaînes. Nous en avons vu un exemple dans le code :

```
MsgBox "le périmètre est de " & Circonference & " mm"
```

Notez que la variable concaténée est une valeur numérique et qu'il y a une conversion implicite vers le type String.

Opérateurs logiques

En Visual Basic, les mêmes opérateurs servent pour la logique et la logique binaire. Ce n'est pas forcément une bonne chose en terme de lisibilité mais avec l'habitude on évite les erreurs que cela pourrait entraîner. Commençons avec la logique standard, nous verrons la logique binaire juste après.

Les opérateurs logiques s'utilisent toujours sous la forme :

résultat **=** *expression1* **opérateur** *expression2*

A l'exception de l'opérateur Not. Expression1 et expression2 doivent donc renvoyer des valeurs booléenne (vrai/faux) et le résultat en sera une aussi.

Opérateur And

Vérifie que les deux expressions sont vraies.

Expression1	Expression2	Résultat
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Faux

Si une expression renvoie Null, le résultat sera Null si l'autre expression est vraie, faux sinon.

Opérateur Or

Vérifie qu'au moins une des deux expressions sont vraies.

Expression1	Expression2	Résultat
Vrai	Vr	Vrai
Vra	Faux	Vrai
Fau	Vrai	Vrai
Faux		Faux

Si une expression renvoie Null, le résultat sera Null si l'autre expression est fausse, vrai sinon.

Opérateur Eqv

Vérifie l'équivalence logique c'est-à-dire que les deux expressions renvoient le même résultat.

Expres	Expression2	Résultat
Vra	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Faux
Faux	Faux	Vrai

Si une expression renvoie Null, le résultat sera Null.

Opérateur XOr

Vérifie l'exclusion logique c'est-à-dire que les deux expressions renvoient un résultat différent.

Expression1	Expression2	Résultat
Vrai	Vrai	Faux
Vrai	Faux	Vrai
Faux	Vrai	Vrai
Faux	Faux	Faux

Si une expression renvoie Null, le résultat sera Null.

Opérateur Imp

Vérifie l'implication logique c'est-à-dire que si l'expression1 est vraie, l'expression2 doit l'être aussi.

Expression1	Expression2	Résultat
Vrai	Vrai	Vrai
Vrai	Faux	Faux
Faux	Vrai	Vrai
Faux	Faux	Vrai
Null	Vrai	Vrai
Null	Faux	Null
Vrai	Null	Null
Faux	Null	Faux
Null	Null	Null

Opérateur Not

Applique la négation logique, c'est-à-dire inverse le résultat d'une expression. De la forme :
résultat **= Not** *expression*

Combinaisons d'opérateur

Ces opérateurs peuvent être combinés dans une expression, cependant il existe des priorités dans leur évaluation. Lorsqu'il y a plusieurs types d'opérateurs, ils sont évalués dans l'ordre :
Opérateurs arithmétiques → opérateurs de comparaison → opérateurs logiques
Au sein d'une même famille, les priorités suivent l'ordre suivant :

Arithmétique	Comparaison	Logique
Élévation à une puissance (^)	Égalité (=)	Not
Négation (–)	Inégalité (<>)	And
Multiplication et division (*, /)	Infériorité (<)	Or
Division d'entiers (\)	Supériorité (>)	Xor
Modulo arithmétique (Mod)	Infériorité ou égalité (<=)	Eqv
Addition et soustraction (+, –)	Supériorité ou égalité (>=)	Imp
Concaténation de chaînes (&)		

- Lorsque deux opérateurs ont la même priorité, ils sont traités dans l'ordre d'apparition de la gauche vers la droite.
- Les parenthèses permettent de modifier la priorité
- Toute l'expression est toujours évaluée

Opérateur d'affectation, =

A la différence d'autres langages, VBA utilise le même opérateur pour l'affectation et pour l'égalité. Si le contexte permet généralement de savoir quel est son rôle dans une ligne de code donnée, il peut quand même il y avoir des ambiguïtés dans certains cas. Imaginons le code suivant :

```
Sub OperateurEgal()

Dim Var1 As Integer, Var2 As Integer, Var3 As Integer

Var1 = 10
Var3 = Var2 = Var1
Debug.Print Var3

End Sub
```

Contrairement aux apparences, ce code ne veut pas dire "mettre var2 et var3 à 10 mais "mettre dans var3 le résultat de l'expression var2 = var1". Dès lors, on pourrait s'attendre à obtenir une erreur de type puisque le résultat est une valeur booléenne et que var3 est de type entier. Comme nous l'avons dit précédemment, il s'agit d'un cas où VBA va convertir implicitement le type pour le faire coïncider avec le type de la variable var3. Pour que ce code soit écrit correctement il faudrait typer la variable comme valeur booléenne.

Attention, il existe aussi une Instruction **Set** pour affecter des objets dans des variables. Cette instruction est obligatoire pour que l'affectation soit correcte.

Logique binaire

La logique binaire, qu'on retrouve parfois nommée sous l'élégante traduction de "masque de bits" (bitmask), consiste à utiliser la représentation binaire des nombres pour véhiculer plusieurs réponses booléennes.

Regardons les valeurs des seize premiers nombres entiers et leurs représentations binaires. On peut imaginer ses quatre bits comme quatre réponses booléennes. Par exemple comme l'encadrement d'une cellule ou 1 indiquerait que la bordure est tracée et zéro non, et où les bits 1 2 3 et 4 correspondrait respectivement aux bordures gauche haute droite et basse.

Valeur décimale		1	2	3	4
0	→	0	0	0	0
1	→	0	0	0	1
2	→	0	0	1	0
3	→	0	0	1	1
4	→	0	1	0	0
5	→	0	1	0	1
6	→	0	1	1	0
7	→	0	1	1	1
8	→	1	0	0	0
9	→	1	0	0	1
10	→	1	0	1	0
11	→	1	0	1	1
12	→	1	1	0	0
13	→	1	1	0	1
14	→	1	1	1	0
15	→	1	1	1	1

Vous noterez aussi que chaque puissance de deux donne un résultat qu'on ne peut obtenir par addition des nombres inférieurs. Pour travailler en représentation binaire, VBA utilise une énumération de puissance de 2. Les énumérations VBA se déclarent avec l'instruction Enum. Pour manipuler ses valeurs, on peut utiliser indifféremment les opérateurs logiques ou les opérations décimales.

Regardons l'exemple suivant :

```
Public Enum Bordure
    Gauche = 1 '2^0
    Haute = 2 '2^1
    droite = 4 '2^2
    basse = 8 '2^3
End Enum

Public Sub Encadrement()
 Dim Encadre As Bordure
    'ajoute la bordure de gauche
    Encadre = Encadre Or Gauche
    'ajoute la bordure haute
    Encadre = Encadre + Haute
    'ajoute la bordure droite
    Encadre = Encadre + 4
    'ajoute la bordure basse
    Encadre = Encadre Xor basse
    Debug.Print Encadre
End Sub
```

La première partie du code définit l'énumération. Dans la procédure encadrement, j'ajoute des valeurs soit par addition soit par utilisation des opérateurs logiques. Bien qu'il faille raisonner en terme de masque binaire, la valeur utilisée est un entier. L'affichage de la variable Encadre en fin de procédure renverra la valeur 15.

Comme l'utilisation des opérateurs arithmétiques est "moins souple", on utilise généralement les opérateurs logiques, qui fonctionnent en comparaison de bits. Leur fonctionnement est similaire à la logique standard, c'est à dire de la forme :

résultat **=** *expr1* **opérateur** *expr2*

Les réponses de ses opérateurs sont définies dans le tableau ci-après

Opérateurs	Bit dans expr1	Bit dans expr2	résultat
And	0	0	0
	0	1	0
	1	0	0
	1	1	1
Eqv	0	0	1
	0	1	0
	1	0	0
	1	1	1
Imp	0	0	1
	0	1	1
	1	0	0
	1	1	1
Or	0	0	0
	0	1	1
	1	0	1
	1	1	1
XOr	0	0	0
	0	1	1
	1	0	1
	1	1	0

Attention, les actions ont lieu sur tout le masque ce qui peut produire des résultats surprenant lorsqu'on n'est pas vigilant. Reprenons l'énumération précédente et essayez de trouver le résultat qu'aura la variable Encadre sans exécuter le code.

```
Public Sub ValeurEncadre()

 Dim Encadre As Bordure

    Encadre = Encadre Xor basse
    Encadre = Encadre Or droite
    Encadre = haute Imp Encadre
    Encadre = Encadre And gauche
    Encadre = Not Encadre

End Sub
```

Procédures & fonctions

On appelle procédure (méthode dans le cas des objets) un ensemble d'instructions s'exécutant comme une entité. Tout code exécutable est obligatoirement dans une procédure.

En VBA, une procédure se définit quand on la déclare. La définition est toujours constituée telle que :

- Un mot clé définissant la portée [1]
- L'instruction *Sub*
- Le nom de la procédure
- Éventuellement la liste des arguments

Par exemple, la ligne suivante définie la procédure publique *Encadrement* :

```
Public Sub Encadrement()
```

Lorsqu'une procédure renvoie une valeur, on l'appelle alors fonction. Une fonction se définit comme une procédure sauf qu'on utilise l'instruction *Function* au lieu de *sub*, et qu'on précise généralement le type renvoyé. Si le type n'est pas précisé, la fonction est de type Variant.

La ligne suivante définit la fonction CalculPerimetre qui renvoie un résultat de type décimal.

```
Public Function CalculPerimetre() As Decimal
```

Comme la définition et la déclaration ont lieu en même temps, cette ligne doit être suivie par la ou les lignes de code composant la procédure et finir par l'instruction `End Sub` ou `End Function` selon le cas.

Arguments

Souvent appelés paramètres, les arguments sont des variables (au sens large du terme) que l'on communique à la procédure. Les arguments se déclarent lors de la définition de la procédure. Les paramètres suivent les règles de déclaration suivantes :

- Un mot clé spécifique à la déclaration d'arguments
- Le nom de l'argument
- Son type
- Éventuellement sa valeur

Par exemple la ligne suivante déclare la fonction IntToBin qui convertit un nombre entier dans sa représentation binaire et prends un argument de type entier :

```
Private Function IntToBin(ByVal IntegerNumber As Long) As String
```

ByRef & ByVal

Pour comprendre correctement le type de passage d'un argument, il faut concevoir les représentations d'une variable. Une variable est un emplacement mémoire qui contient quelque chose. Stricto sensu il s'agit donc d'une adresse mémoire. Cependant à chaque instant, elle a aussi une valeur qui elle, est l'information qu'on utilise. Comme la variable est de fait son adresse mémoire, une action en tout point du code modifie la variable. Ce comportement n'est pas forcément souhaitable ou indésirable, tout dépend de ce que l'on veut obtenir.

Si on désire pouvoir modifier la valeur d'un argument au sein d'une fonction et répercuter ces modifications dans le code appelant, on dit qu'on passe la variable par référence, c'est-à-dire que l'argument contient l'adresse de la variable. Si on souhaite que les modifications ne soient pas répercutées, on passe l'argument par valeur, c'est-à-dire en copiant la valeur dans une variable locale. Le code suivant montre bien le principe des types de passage.

```
Private Sub TypePassage(ByRef ParReference As Integer, ByVal ParValeur As
Integer)
   ParReference = 2 * ParReference
   ParValeur = 2 * ParValeur
End Sub
```

[1] Si cette portée est omise, la procédure sera publique si elle est dans un module standard, privée si non.

```
Public Sub Test()

Dim EntierRef As Integer, EntierVal As Integer

EntierRef = 1
EntierVal = 1
TypePassage EntierRef, EntierVal
MsgBox "Par référence " & EntierRef & vbCrLf & "Par valeur " & EntierVal

End Sub
```

Et vous obtiendrez :

Comme vous le voyez, la variable passée par référence a bien été modifiée par la procédure *TypePassage* alors que la variable passée par valeur ne l'a pas été.

Par défaut, c'est-à-dire si vous ne précisez ni l'un ni l'autre, les variables sont passées par référence au nom de l'efficacité puisqu'une adresse ne prend que quatre octets. Ce n'est pas pénalisant en soit, pour peu que vous ayez bien conscience que vous ne devez modifier la variable qu'à bon escient.

De manière générale, on suit les règles suivantes.

- Les valeurs simples (nombres, chaînes, dates…) sont transmises par valeur.
- Les objets et les tableaux sont transmis par référence.
- On utilise plutôt des variables globales pour modifier une valeur qu'un argument passé par référence[2].

Optional

Ce modificateur précise que l'argument est optionnel, c'est-à-dire qu'il peut être transmis comme il peut ne pas l'être. Les règles suivantes s'appliquent :

- Lorsqu'un argument est marqué comme optionnel, tout les arguments qui le suivent dans la déclaration doivent être marqués comme étant optionnels.
- Les arguments optionnels ne peuvent pas être utilisés en même temps que les tableaux d'arguments (ParamArray)
- L'absence du paramètre peut être testé avec la fonction IsMissing si son type n'est pas un type valeur.
- Les arguments optionnels peuvent avoir une valeur par défaut. Dans ce cas IsMissing renvoie toujours False.

Les deux syntaxes qui suivent sont équivalentes :

```
Private Function Arrondir(ByVal Valeur As Single, Optional NombreDeDecimale
As Variant) As Single

    If IsMissing(NombreDeDecimale) Then NombreDeDecimale = 1
    Arrondir = Valeur * (10 ^ NombreDeDecimale)
    Arrondir = Int(Arrondir + 0.49) * (10 ^ -NombreDeDecimale)

End Function
```

Notez que pour que cette fonction fonctionne correctement, l'argument NombreDeDecimale est de type Variant et non de type Integer.

[2] Encore que d'autres auteurs vous diront exactement l'inverse.

```
Private Function Arrondir(ByVal Valeur As Single, Optional NombreDeDecimale
As Integer = 1) As Single

    Arrondir = Valeur * (10 ^ NombreDeDecimale)
    Arrondir = Int(Arrondir + 0.49) * (10 ^ -NombreDeDecimale)

End Function
```

ParamArray

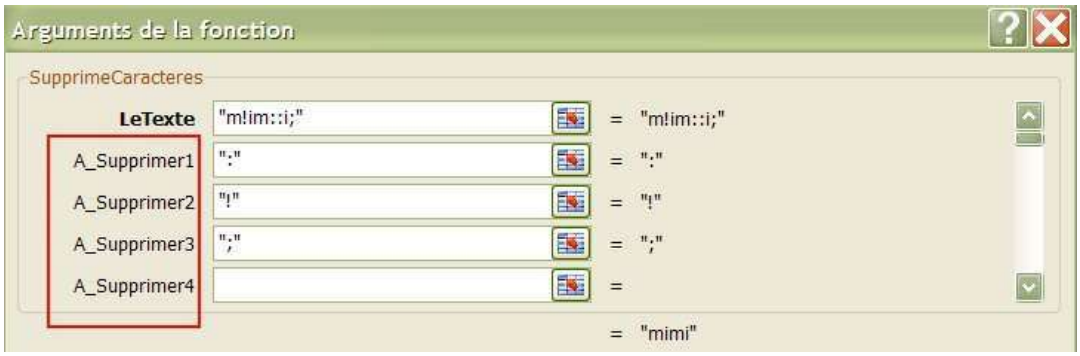
Indique que l'argument est un tableau variant de paramètre. Ce mot clé ne peut être utilisé que sur le dernier argument. Ce mot clé n'est guère utilisé car il présente quelques difficultés de manipulation, car il représente en fait un nombre illimité d'arguments. Prenons l'exemple suivant :

```
Private Function Minimum(ParamArray Elements() As Variant) As Variant

Dim cmpt As Long
If UBound(Elements) > -1 Then 'au moins un argument
    For cmpt = LBound(Elements) To UBound(Elements)
        If IsNumeric(Elements(cmpt)) Then
            If Not IsEmpty(Minimum) Then
                If Minimum > Elements(cmpt) Then Minimum = Elements(cmpt)
            Else
                Minimum = Elements(cmpt)
            End If
        End If
    Next cmpt
End If

End Function

Public Sub Test()

    MsgBox Minimum("toto", -1, True, 124, Range("B2"), 25.471)

End Sub
```

Comme nous le voyons, si la fonction déclare bien un argument de type tableau, l'appel utilise lui une liste d'éléments. Le tableau étant de type variant, les éléments peuvent être de n'importe quel type.

Notez que l'indice du premier argument est toujours 0.

Si vous donnez à l'argument marqué ParamArray un nom de variable finissant par 1 dans une fonction de feuille de calcul, l'assistant de fonctions incrémentera automatiquement chaque nouvel élément :



Arguments nommés ou passage par position

Comme nous venons de le voir, une procédure peut avoir un grand nombre d'arguments. Normalement les arguments sont passés par position, c'est-à-dire dans l'ordre où ils apparaissent dans la déclaration de la procédure. Cependant, ce n'est pas toujours commode de passer de nombreux arguments facultatifs vides uniquement parce que l'on souhaite passer le premier et le dernier.

Imaginons un appel à la fonction VBA Replace dont la définition est de la forme simplifiée :

Function Replace(expression As String, find As String, replace As String, Optional start As Long, Optional count As Long, Optional compare As VbCompareMethod) **As String**

Nous souhaitons passer les trois arguments obligatoires et le dernier argument optionnel qui précise le type de comparaison. Pour réaliser cet appel en passant les arguments par position, nous devons écrire :

MaChaîne=Replace(MaChaîne,",",";",,,VbTextCompare)

C'est-à-dire que nous passons autant de virgule que d'arguments. Pour deux arguments optionnels, c'est encore acceptable, mais s'il y en a 10, c'est illisible.

On peut alors passer les arguments en les nommant. Dès lors, il n'y a pas besoin de respecter l'ordre, ni de passer des emplacements vides pour les arguments optionnels.

Le nom de l'argument est défini par le modèle de la fonction. Ainsi dans la fonction Replace, le premier argument s'appelle 'expression', le deuxième 'find' et ainsi de suite. La valeur est affectée à l'argument à l'aide de l'opérateur particulier := qui n'est utilisé que dans ce cas. Nous pourrions donc écrire un appel identique au précédent sous la forme :

MaChaîne=Replace(expression := MaChaîne, compare := VbTextCompare)

Instructions et règles d'appel

Pour appeler une fonction ou une procédure depuis un autre point du code, il faut d'abord et avant tout que sa portée le permette, c'est-à-dire qu'elle soit visible depuis le point d'appel. Autrement dit, vous ne pouvez pas appeler une fonction déclarée *Private* dans un formulaire à partir d'une procédure d'un module standard.

L'appel classique consiste à écrire le nom de la procédure suivi de la liste des arguments en suivant les règles suivantes :

- Si une procédure est appelée, on tape son nom suivi de la liste des arguments séparés par des virgules.
- Si une fonction est appelée on tape son nom suivi de la liste des arguments, entre parenthèses, séparés par des virgules.

On utilise parfois l'instruction Call pour appeler une procédure ou une fonction dont on ne souhaite pas récupérer la valeur de retour. Dès lors qu'on utilise Call, la liste des arguments doit être mise entre parenthèse. Nous voyons un exemple ci-après :

```
Private Sub ExempleSub(Arg1 As Integer)
    MsgBox Arg1
End Sub

Private Function ExempleFunction(Arg1 As Integer) As String
    ExempleFunction = CStr(Arg1)
End Function

Public Sub TestAppel()

    'appel classique
    ExempleSub 12
    'appel avec call
    Call ExempleSub(12)
    'appel de fonction
    MsgBox ExempleFunction(12)
    'valeur de retour ignorée
    Call ExempleFunction(12)
End Sub
```

Valeur retournée

La différence entre une fonction et une procédure repose sur le fait que la fonction renvoie une valeur typée. Pour définir la valeur, on utilise une variable implicite qui est le nom de la fonction. Autrement dit, vous devrez avoir quelque part dans la fonction soit :

```
NomDeLaFonction = Valeur
```

Ou

```
Set NomDeLaFonction = Valeur
```

Par exemple:

```
Private Function Conv2Fahrenheit(ByVal TCelsius As Single) As Single
    Conv2Fahrenheit = 1.8 * TCelsius + 32
End Function

Private Function NumericRange(Feuille As Worksheet) As Range
    Set NumericRange = Feuille.Cells.SpecialCells(xlCellTypeConstants,
xlNumbers)
End Function

Sub AppelFonction()

    Dim PlageNumerique As Range, TempC As Single, TempF As Single

    TempC = 25
    TempF = Conv2Fahrenheit(TempC)
    Set PlageNumerique = NumericRange(ActiveSheet)
    PlageNumerique.Value = TempF

End Sub
```

Les fonctions peuvent évidemment avoir plusieurs points de sorties, logiques ou forcés à l'aide de l'instruction "Exit". Les règles de la bonne programmation veulent qu'il y ait une affectation de valeur à chaque point de sortie.

```
Private Function RenvoieSigne(ByVal Valeur As Single) As Integer

    Select Case Valeur
        Case Is < 0
            RenvoieSigne = -1

        Case 0 'ce cas est implicitement vrai
            RenvoieSigne = 0

        Case Is > 0
            RenvoieSigne = 1

    End Select

End Function
```

Cette fonction à trois points de sortie. Traiter le cas 0 n'est utile qu'en terme de lisibilité et de bonne programmation puisque de fait si valeur vaut 0, la fonction RenvoieSigne vaudra 0 si nous ne lui affectons pas explicitement une valeur différente (0 étant la valeur d'initialisation de la variable RenvoieSigne).

```
Private Function EstVide(ByRef MaPlage As Range) As Boolean

Dim MaCellule As Range
    For Each MaCellule In MaPlage.Cells
        If Not IsEmpty(MaCellule) Then
            EstVide = False 'implicitement vrai
            Exit Function
        End If
    Next
    EstVide = True

End Function
```

Cette fonction à deux points de sortie (un point forcé). Là encore, la ligne EstVide = False n'est utile que pour des raisons de bonne programmation puisque EstVide vaut False sauf affectation explicite de la valeur True.

Les objets

Un objet est un ensemble code / données se comportant comme une entité unique et cohérente. Si VBA n'est pas un langage objet, le modèle Excel en est entièrement composé, il va donc falloir nous familiariser avec leurs manipulations.

Un objet expose une interface constituée de méthodes, de propriétés et d'évènements, tout cela compose les membres de l'objet. Oublions pour l'instant les évènements.

Un objet possède donc des propriétés définissant son état et des fonctionnalités qui sont ses méthodes. Il possède par ailleurs un type issu de la classe qui le définit. Une classe est l'ensemble du code qui définit un type d'objet. Pour faire une analogie classique, la classe est la recette de cuisine et l'objet le plat. L'objet étant l'instance d'une classe, il doit être instancié, mais cela n'est pas forcément dû à votre code. Revenons-en à Excel VBA. Un certain nombre d'objet que vous allez manipuler seront directement ou implicitement instanciés (créés) par l'application Excel.

Lorsqu'on doit instancier un objet par le code, on utilise le mot clé "***New***" ou dans certains cas une méthode fournit par un autre objet.

Nous reviendrons plus loin sur la conception des objets, concentrons nous pour l'instant sur leur manipulation. Pour pouvoir utiliser un objet, vous devez accéder soit à l'objet directement, soit à une variable y faisant référence. A la différence des valeurs, on doit utiliser le mot clé "***Set***" pour affecter une variable objet. Notez que dans certains, cas le moment de l'affectation peut avoir son importance.

Pour utiliser un membre de l'objet on utilise l'opérateur "." (Point). La nature du membre va définir la syntaxe à utiliser. Les propriétés attendent et/ou renvoient des valeurs, les méthodes attendent éventuellement des arguments et renvoient éventuellement une valeur. Pour faire une analogie, les propriétés se traitent comme des variables et les méthodes comme des appels de procédures / fonctions.

Comme un petit exemple est souvent plus efficace qu'un long discours, regardons l'exemple de code commenté ci-dessous.

```
Sub DemoObjet()

Dim Classeur As Workbook

    Set Classeur = ActiveWorkbook
    'valorise la propriété Date1904 du classeur
    ActiveWorkbook.Date1904 = True
    'lit la même propriété à l'aide de la variable
    MsgBox Classeur.Date1904
    'utilise une méthode de la variable objet
    Classeur.Protect Password:="MotDePasse"
    'utilise une méthode de l'instance
    ActiveWorkbook.Unprotect "MotDePasse"
    'une propriété peut être un objet
    'lit une propriété d'un objet contenu par l'objet
    Dim NomFeuille1 As String
    NomFeuille1 = Classeur.Worksheets(1).Name
    'utilise une méthode d'un objet contenu
    ActiveWorkbook.ActiveSheet.Delete
    'tend un filochon à bourricot
    Workbooks.Add
    MsgBox ActiveWorkbook.Sheets.Count & vbNewLine & Classeur.Sheets.Count

End Sub
```

Notez que les variables objet se déclarent comme n'importe quelle variable.

Reprenons quelques uns des points que nous avons vus illustré par ce code. Tout d'abord, si le type de l'objet est "Workbook" (classeur) il n'existe pas d'instance Workbook. De fait, ActiveWorkbook est une référence créée par Excel du classeur actif. Vous noterez aussi que nous pouvons agir indifféremment sur l'instance ou sur la variable faisant référence à celle-ci.

Nous voyons aussi qu'un objet peut avoir comme membres des propriétés étant des objets, qui eux-mêmes peuvent avoir des propriétés étant des objets et ainsi de suite. On appelle cela un modèle objet hiérarchique et c'est le fondement du modèle objet d'Excel.

Les propriétés ont toujours un type, objet ou valeur, les arguments et les valeurs renvoyées par les méthodes également. Autrement dit, il n'existe pas de grandeurs sans type.

J'ai ajouté à la fin du code deux lignes de code présentant un petit piège. Jusque la, la variable Classeur et l'objet ActiveWorkbook faisait référence à la même chose. L'appel de la méthode Workbooks.Add demande à Excel de créer un nouveau classeur. Ce nouveau classeur devient le classeur actif par défaut, dès lors, Excel fait référence à ce nouveau classeur dans l'objet ActiveWorkbook. Par contre notre variable classeur fait toujours référence à l'ancien classeur actif, il n'y a donc plus équivalence entre la variable et l'objet ActiveWorkbook.

Restons en là pour l'instant, nous reviendrons beaucoup plus en détails sur les objets dans la suite de ce cours.

Les tableaux

On appelle tableau, un ensemble d'éléments, indexés séquentiellement, ayant le même type de données. Un tableau peut posséder jusqu'à 60 dimensions en Visual Basic. Les déclarations des tableaux suivent les règles suivantes :

Les tableaux statiques doivent définir leurs nombres d'éléments dans chaque dimension lors de leur déclaration :

```
Dim TabEntier(15) As Integer
Dim TabString(10, 10) As String
```

En VBA, les tableaux peuvent avoir par défaut comme plus petit indice, 0 ou 1.

On définit cela en précisant l'Option Base en tête de module. Option Base 0 veut dire que le plus petit indice des tableaux dont la limite inférieure n'est pas explicitement définie est 0. Autrement dit :

```
Option Explicit
Option Base 1

Private TabNom(10) As String
```

Signifie que TabNom va de 1 à 10 alors que si l'instruction Option Base valait 0 ou si elle était omise, il irait de 0 à 9.

VBA permet la déclaration explicite des limites de chaque dimension à l'aide de la clause "To". Par exemple :

```
Dim TabBool(1 To 10) As Boolean
Dim TabSingle(5 To 15, 3 To 8, 2 To 7) As Single
```

Tous les éléments d'un tableau sont de même type. Cependant comme un tableau peut être de type Variant, un tableau peut contenir tout type de données, y compris des objets ou d'autres tableaux.

Les tableaux dynamiques se déclarent sans préciser les dimensions mais avec les parenthèses.

```
Dim TabDynamique() As Single
```

On utilise l'instruction ReDim pour modifier la taille d'un tableau dynamique, que ce soit le nombre d'éléments ou le nombre de dimensions.

```
ReDim TabDynamique(1 To 10)
ReDim TabDynamique(1 To 5, 1 To 5)
```

Le redimensionnement engendre la perte des données contenues. Pour conserver ses données, on utilise le mot clé Preserve.

```
Dim TabDynamique() As Single

ReDim TabDynamique(1 To 5, 1 To 5)
TabDynamique(1, 1) = 2.54
ReDim Preserve TabDynamique(1 To 5, 1 To 6)
MsgBox TabDynamique(1, 1)
```

L'utilisation du mot clé Preserve fige le nombre de dimensions et la limite inférieure de chacune d'entre elle, autrement dit, les trois dernières lignes suivantes déclencheraient une erreur :

```
ReDim TabDynamique(1 To 5, 1 To 5)
ReDim Preserve TabDynamique(1 To 5, 1 To 6)
ReDim Preserve TabDynamique(1 To 6, 1 To 5)
ReDim Preserve TabDynamique(1 To 5, 2 To 5)
ReDim Preserve TabDynamique(1 To 6)
```

On lit ou écrit dans un élément du tableau en nommant le tableau avec le(s) indice(s) de la case comme montré dans l'exemple précédent. Tout appel d'un tableau avec un indice en dehors de la plage déclenchera une erreur récupérable.

Instructions et fonctions spécifiques

L'instruction **Erase** permet d'effacer un tableau. Dans le cas des tableaux fixes, les valeurs sont réinitialisées mais le tableau garde les dimensions qui sont définies, dans le cas des tableaux dynamiques, le tableau est réinitialisé (perte des valeurs et des dimensions).

La notation est :

Erase *Tableau*

La fonction **Array** permet d'initialiser une variable de type Variant comme un tableau en lui passant directement des éléments du tableau. La limite est fixée par la valeur d'Option Base, sauf si vous utilisez la syntaxe *VBA.Array*. Dans ce cas, l'indice de la limite basse sera forcé à 0. On obtient forcément un tableau unidimensionnel avec cette fonction.

Il existe deux fonctions permettant de détecter les limites des tableaux, **LBound** et **Ubound**.

De la forme :

LBound(*arrayname* [**,** *dimension As Integer*]) **As Long**

UBound(*arrayname* [**,** *dimension As Integer*]) **As Long**

Où *arrayname* est le nom de la variable tableau et *dimension* le numéro facultatif de la dimension dont on cherche les limites. Lorsque dimension est omis, la valeur de la première dimension est renvoyée.

```
Option Base 0

Public Sub TestTableau()

Dim TabVar1 As Variant

    TabVar1 = Array(1, 2, 3, 4, 5)
    Debug.Print "limite inf " & LBound(TabVar1) & vbNewLine & "limite sup "
& UBound(TabVar1)
    'limite inf 0
    'limite sup 4
    Erase TabVar1
    ReDim TabVar1(0 To 2, 1 To 5)
    Debug.Print "limite inf dimension 2 " & LBound(TabVar1, 2)
    'limite inf dimension 2 1

End Sub
```

Les blocs

L'instruction **With…End With** est l'instruction de bloc Visual Basic. L'instruction de bloc permet de faire référence à un objet une seule fois pour tout le bloc de ligne de codes qu'il contient. Ceci permet une augmentation de vitesse du code puisque la référence n'est résolue qu'une fois et une meilleure lisibilité. Par exemple :

```
Dim Valeur As Variant

With Range("B2")
    .Font.ColorIndex = 7
    .Interior.ColorIndex = 5
    Valeur = .Value
    .BorderAround ColorIndex:=3, Weight:=xlThick
    .FormulaLocal = "=Somme(A1:A10)"
End With
```

Il est possible d'imbriquer les blocs With dans le sens du modèle hiérarchique :

```
With Workbooks("Classeur1").Worksheets("Feuil1").Cells(2, 2)
    .FormulaLocal = "=Somme(A1:A10)"
    With .Font
        .Name = "Arial"
        .Bold = True
        .Size = 8
    End With
    With .Interior
        .ColorIndex = 5
        .Pattern = xlPatternCrissCross
    End With
End With
```

Structure décisionnelle

VBA utilise plusieurs formes de structures décisionnelles. On appelle structure décisionnelle la construction de code permettant de tester une ou plusieurs expressions afin de déterminer le code à exécuter.

Les structures compactes

Les structures compactes sont des fonctions VBA retournant une valeur en fonction d'un test.

Immediate If ⇨ IIf

De la forme **IIf(*expr*, *truepart*, *falsepart*)**

Où truepart et falsepart sont des valeurs ou des expressions renvoyées. Par exemple :

```
Private Function IsNothing(ByRef Objet As Object) As Boolean

    IsNothing = IIf(Objet Is Nothing, True, False)

End Function
```

Il est possible d'imbriquer ce type de structure, bien que la lisibilité en pâtisse rapidement. Cette imbrication peut être hiérarchique ou indépendante :

```
'imbrication hiérarchique
Private Function Signe(ByVal Valeur As Double) As Integer

    Signe = IIf(Valeur > 0, 1, IIf(Valeur < 0, -1, 0))

End Function

'imbrication indépendante
Private Function Max2(ByVal Valeur1 As Variant, ByVal Valeur2 As Variant)
As Variant

    Max2 = IIf(IIf(IsNumeric(Valeur1), Valeur1, Empty) <
IIf(IsNumeric(Valeur2), Valeur2, Empty), Valeur2, Valeur1)

End Function
```

Attention, Visual Basic évalue toujours toutes les expressions de la structure même la partie qui n'est pas renvoyée. Si celle-ci contient une erreur, elle se produira même si l'expression ne devrait pas être renvoyée. Dans l'exemple suivant, la fonction lèvera une erreur si l'argument vaut Nothing car Nothing.Value est une erreur même si cette partie de l'expression ne doit pas être évaluée.

```
Private Function IIFBug(ByVal Arg1 As Range) As Integer

    IIFBug = IIf(Arg1 Is Nothing, 0, Arg1.Value)

End Function
```

Choose

Renvoie une valeur d'une liste de choix en fonction de l'index passé en premier argument. De la forme :

Choose(Index, Choix$_1$, Choix$_2$, …., Choix$_n$)

```
Private Function EqvColorIndex(ByVal Index As Integer) As String

    EqvColorIndex = Choose(Index, "noir", "blanc", "rouge", "vert", "bleu",
"jaune")

End Function
```

Index doit forcément être compris entre 1 et le nombre de choix de la liste, sinon Choose renverra la valeur Null.

```
Private Function EqvColorIndex(ByVal Index As Integer) As String
    Dim Reponse As Variant
    Reponse = Choose(Index, "noir", "blanc", "rouge", "vert", "bleu",
"jaune")
    EqvColorIndex = IIf(IsNull(Reponse), "index en dehors des limites",
Reponse)
End Function

Private Sub test2()
    MsgBox EqvColorIndex(7)
End Sub
```

Si Index est un nombre décimal, il est arrondi avant l'évaluation, dans notre exemple précédent, MsgBox EqvColorIndex(2.8) renverra "rouge".

Attention, là encore tous les choix seront évalués. L'exemple suivant affichera la liste des couleurs dans des boîtes de dialogue :

```
Private Sub EqvColorIndex(ByVal Index As Integer)

    Choose Index, MsgBox("noir"), MsgBox("blanc"), MsgBox("rouge"),
MsgBox("vert"), MsgBox("bleu"), MsgBox("jaune")

End Function
```

Switch

Cette fonction renvoie une valeur ou une expression fonction de l'expression qui lui est associée dans une liste d'expression. On la rencontre rarement car elle est assez souvent mal comprise.

Sa forme générique est :

Switch(*expr-1*, *value-1*[, *expr-2*, *value-2* … [, *expr-n*,*value-n*]])

ou autrement dit :

Switch(expression 1 à évaluer, expression renvoyée si expression 1 est vrai, expression 2 à évaluer, expression renvoyée si expression 2 est vrai, etc…)

Commençons par un exemple simple d'évaluation de signe similaire à la fonction écrite avec des IIf imbriqués :

```
Private Function Signe2(ByVal Valeur As Double) As Integer

    Signe2 = Switch(Valeur < 0, -1, Valeur = 0, 0, Valeur > 0, 1)

End Function
```

Dans ce cas, la fonction va évaluer les expressions de la gauche vers la droite et renvoyer la valeur associée dès lors qu'une des expressions est vraie. L'ordre d'évaluation a son importance puisque c'est la première expression vraie qui définira la valeur renvoyée. Ainsi la fonction suivante renverrait la même chose que la précédente quand bien même les deux dernières expressions peuvent être vraies si Valeur vaut 0 :

```
Private Function Signe2(ByVal Valeur As Double) As Integer
    Signe2 = Switch(Valeur < 0, -1, Valeur = 0, 0, Valeur >= 0, 1)
End Function
```

Switch renverra Null si aucune expression n'est vraie, par exemple :

```
Public Function SousType1(ByVal Valeur As Variant) As String

Dim Reponse As Variant
Reponse = Switch(IsArray(Valeur), "Tableau", IsEmpty(Valeur), "Vide",
IsDate(Valeur), "Date", IsNumeric(Valeur), "Nombre")
SousType1 = IIf(IsNull(Reponse), "Chaîne", Reponse)

End Function
```

Il est possible d'imbriquer les fonctions Switch :

```
Public Function SousType1(ByVal Valeur As Variant) As String

Dim Reponse As Variant
Reponse = Switch(IsArray(Valeur), "Tableau", IsEmpty(Valeur), "Vide",
IsDate(Valeur), "Date", IsNumeric(Valeur), Switch(Valeur - Int(Valeur) <>
0, "Decimal", Valeur - Int(Valeur) Mod 1 = 0, "Entier"))
SousType1 = IIf(IsNull(Reponse), "Chaîne", Reponse)

End Function

Private Sub test2()

    MsgBox SousType1(14.3)'renverra 'Decimal

End Sub
```

Là encore, toutes les expressions quelles soient de test ou renvoyées seront évaluées. Dès lors la fonction précédente renverra une erreur d'incompatibilité de type pour toute données non numériques passées comme argument puisque Int() attend forcément un nombre.

Quoique très pratique, ces structures compactes ont donc deux inconvénients majeurs, leur faible lisibilité et l'évaluation de tous les cas mêmes lorsqu'ils sont exclus par le test. Pour éviter ces inconvénients, on travaille plutôt avec les structures éclatées suivantes.

If … Then… Else

La structure If – Then – Else admet deux syntaxes, la syntaxe linéaire et la syntaxe de blocs.

Dans la syntaxe linéaire, toute l'instruction tient sur une ligne et il n'y a pas d'instruction de fin de blocs, la forme en est :

If Test **Then** Traitement **Else** Traitement

Dans la syntaxe de bloc, les traitements sont séparés des instructions du bloc et le bloc se termine forcément par l'instruction "End If". La structure est donc de la forme

If *Test* **Then**
    Traitement
Else
    Traitement
End If

Par exemple

```
Private Function PremierLundiDuMois(ByVal Mois As Integer, ByVal Annee As
Integer) As Date
    Dim PremierDuMois As Date, NumJour As Integer

    'syntaxe en ligne
    If Mois < 1 Or Mois > 12 Then Err.Raise 5, "Mois", "La valeur du mois
est un entier compris entre 1 et 12"
    PremierDuMois = DateSerial(Annee, Mois, 1)
    NumJour = Weekday(PremierDuMois, vbMonday)
    'syntaxe en bloc
    If NumJour = 1 Then
        PremierLundiDuMois = PremierDuMois
    Else
        PremierLundiDuMois = DateAdd("d", 8 - Weekday(PremierDuMois,
vbMonday), PremierDuMois)
    End If

End Function
```

Avec la syntaxe en bloc, les traitements peuvent contenir plusieurs lignes de code. Il est possible d'imbriquer des structures If. Notez que quelle que soit la syntaxe choisie, le bloc Else est toujours facultatif.

```
Private Function Max(ParamArray ListeValeur() As Variant) As Variant

    Dim compteur As Long
    If UBound(listvaleur) > -1 Then
        For compteur = LBound(ListeValeur) To UBound(ListeValeur)
            If IsNumeric(ListeValeur(compteur)) Then
                If Max < ListeValeur(compteur) Then
                    Max = ListeValeur(compteur)
                End If
            End If
        Next compteur
    End If

End Function
```

N.B : L'indentation du code n'est pas obligatoire, mais comme elle permet un plus grand confort de lecture, elle est chaudement recommandée.

L'utilisation de la syntaxe de bloc permet aussi de scinder les tests et les expressions. Ainsi la fonction suivante équivalente à la syntaxe "IIfBug" que nous avons vu plus haut fonctionnera sans problème, puisque seul le code du bloc concerné sera évalué :

```
Private Function IfCorrige(ByVal Arg1 As Range) As Integer

    If Arg1 Is Nothing Then
        IfCorrige = 0
    Else
        IfCorrige = Arg1.Value
    End If

End Function
```

ElseIf…Then

Le jeu d'instruction ElseIf…Then permet de gérer des tests consécutifs dans un bloc If. On peut ajouter autant de bloc ElseIf que l'on veut mais aucun ne peut apparaître après le bloc Else. Dès qu'une expression ElseIf est évaluée Vrai, le groupe de traitement du bloc est exécuté et le bloc If s'achève.

```
Public Function SousType2(ByVal Valeur As Variant) As String

If IsArray(Valeur) Then
    SousType2 = "Tableau"
ElseIf IsEmpty(Valeur) Then
    SousType2 = "Vide"
ElseIf IsDate(Valeur) Then
    SousType2 = "Date"
ElseIf IsNumeric(Valeur) Then
    If Valeur - CInt(Valeur) = 0 Then
        SousType2 = "Entier"
    Else
        SousType2 = "Decimal"
    End If
ElseIf IsError(Valeur) Then
    SousType2 = "Erreur"
ElseIf IsNull(Valeur) Then
    SousType2 = "Null"
ElseIf Valeur Is Nothing Then
    SousType2 = "Nothing"
Else
    SousType2 = "Chaîne"
End If

End Function
```

Select Case

Cette syntaxe est similaire au traitement précèdent puisqu'il s'agit d'une suite d'évaluation de test déterminant le code à exécuter.

De la forme :

Select Case *ExpressionTest*
  Case *ExpressionList*
    traitement
  Case *ExpressionList*
    traitement
  Case Else
    traitement
End Select

Dans sa forme la plus usuelle, on compare une expression ou une variable à une suite de valeurs :

```
Private Function EqvColorIndex2(ByVal Index As Integer) As String

    Select Case Index
        Case 1
            EqvColorIndex2 = "noir"
        Case 2
            EqvColorIndex2 = "blanc"
        Case 3
            EqvColorIndex2 = "rouge"
        Case 4
            EqvColorIndex2 = "vert"
        Case 5
            EqvColorIndex2 = "bleu"
        Case 6
            EqvColorIndex2 = "jaune"
    End Select

End Function
```

On peut proposer plusieurs valeurs dans l'instruction Case, soit sous forme de liste en séparant les valeurs par des virgules, soit sous forme de plage à l'aide du mot clé *To*, soit avec des opérateurs de comparaison et le mot clé *Is*. On peut combiner ces valeurs avec des virgules.

```
Private Sub SyntaxeCase(ByVal Arg1 As Integer)

Select Case Arg1
    Case 0
        Debug.Print "Arg1 est nulle"

    Case 1, 2, 3
        Debug.Print "Arg1 vaut 1 ou 2 ou 3"

    Case 4 To 10
        Debug.Print "Arg1 est compris entre 4 et 10 bornes incluses"

    Case Is < 100
        Debug.Print "Arg1 est inférieure à 100"

    Case 100, 200 To 300, Is > 1000
        Debug.Print "Arg1 vaut 100, ou est compris dans l'interval
[200,300] ou est supérieur à 1000"

    Case Else
        Debug.Print "tous les autres cas"
End Select

End Sub
```

Vous noterez que la syntaxe '*Case Else*' permet d'exécuter un traitement si aucun des cas précédent n'a été vrai.

Lorsqu'on utilise des fonctions booléennes pour les tests ou lorsque les tests ne sont pas liés, on utilise généralement la syntaxe renversée **Select Case True**.

```
Private Function SousType3(ByVal Valeur As Variant) As String

Select Case True
    Case IsArray(Valeur)
        SousType2 = "Tableau"

    Case IsEmpty(Valeur)
        SousType2 = "Vide"

    Case IsDate(Valeur)
        SousType2 = "Date"

    Case IsNumeric(Valeur) And Valeur - CInt(Valeur) = 0
        SousType2 = "Entier"

    Case IsNumeric(Valeur) 'utilise la discrimination du test précédent
        SousType2 = "Decimal"

    Case IsError(Valeur)
        SousType2 = "Erreur"

    Case IsNull(Valeur)
        SousType2 = "Null"

    Case Valeur Is Nothing
        SousType2 = "Nothing"

    Case Else
        SousType2 = "Chaîne"

End Select

End Function
```

Pour le cas ou la variable est de sous type décimal, j'ai utilisé un test dépendant de la non véracité du test précédent. En effet, si la variable est un entier, le cas précédent sera vrai et le cas décimal ne sera pas évalué, il n'y a donc bien qu'une valeur décimale qui entrera dans le second *IsNumeric*.

Les blocs Select Case peuvent être imbriqués.

Cette hiérarchie des tests est importante à bien comprendre à plus d'un niveau. D'abord car la discrimination peut avoir son importance sur la qualité de fonctionnement du code, ensuite parce qu'il peut s'agir d'un élément d'optimisation important.

Les boucles

Les boucles ou traitements itératifs se séparent en deux familles, les traitements à nombres d'itérations finis et les boucles à test. Toutes les boucles VBA tolèrent la clause de sortie *Exit*.

For…Next

De la forme :

For *compteur* = *début* **To** *fin* [**Step** *pas*]
[*traitement*]
[**Exit For**] (clause de sortie)
[*traitement*]
Next [*compteur*]

Un exemple classique serait :

```
Private Function Factorielle(ByVal Arg As Integer) As Long

    Dim compteur As Long
    Factorielle = 1
    For compteur = 1 To Arg Step 1
        Factorielle = Factorielle * compteur
    Next compteur

End Function
```

Lorsque le pas vaut 1 comme dans cet exemple, il est généralement omis et on trouvera la notation

```
    For compteur = 1 To Arg
        Factorielle = Factorielle * compteur
    Next compteur
```

Nous pouvons parcourir une boucle en inversant le minimum et le maximum et en déclarant un pas négatif.

```
    For compteur = Arg To 1 Step -1
        Factorielle = Factorielle * compteur
    Next compteur
```

Notez bien que dans ce cas, vous ne pouvez pas omettre le pas sans quoi la boucle ne sera pas exécutée.

Vous pouvez utiliser des expressions en lieu et place de valeur pour les termes début, fin et pas, mais ces expressions ne seront évaluées qu'une fois lors de l'entrée dans la boucle. Autrement dit, si une des expressions peut renvoyer une valeur différente alors que le code est déjà entré dans la boucle, celle-ci ne sera pas prise en compte. Regardons l'exemple suivant :

```
Public Sub TestFor()

Dim compteur As Long

    compteur = 6
    For compteur = 1 To 4 * compteur Step compteur \ 3
        MsgBox compteur
    Next compteur

End Sub
```

Ce code va afficher tous les nombres impairs compris entre 1 et 24. Lors de l'entrée dans la boucle, comme 'compteur' vaut 6, la ligne sera interprétée comme :

```
For compteur = 1 To 24 Step 2
```

Lors de la première itération, 'compteur' valant alors 3, on pourrait penser que la boucle deviendrait :

```
For compteur = 1 To 12 Step 1
```

Mais comme les expressions ne sont évaluées que lors de l'entrée dans la boucle et non lors de chaque itération, c'est bien la première interprétation qui prévaudra. Notez qu'il faut éviter ce type de syntaxe, par convention, la variable de décompte ne doit pas apparaître dans les expressions de définition de la boucle.

Par contre, vous ne devez en aucun cas affecter une valeur à la variable de décompte au sein de la boucle, sous peine de risquer la création d'une boucle infinie comme dans l'exemple suivant :

```
Public Sub TestFor()

Dim compteur As Long

    For compteur = 1 To 4
        compteur = IIf(compteur > 3, 1, compteur)
    Next compteur

End Sub
```

On utilise parfois la clause de sortie 'Exit For' pour sortir prématurément de la boucle. Avant d'utiliser celle-ci, il convient de vérifier qu'un autre type de boucle ne serait pas plus appropriée. Si telle n'est pas le cas, vous pouvez vérifier la cause de la sortie en testant la variable de décompte.

N.B : La clause de sortie positionne le curseur d'exécution sur la ligne qui suit l'instruction Next de la boucle For.

```
Public Sub EspaceConsecutif()

    If TestSortie("Ai-je deux espaces consécutifs  dans la chaîne") Then
        Debug.Print "oui"
    End If

End Sub

Private Function TestSortie(ByVal Phrase As String) As Boolean

Dim compteur As Long, TabSplit() As String

    TabSplit = Split(Phrase, " ")
    For compteur = 0 To UBound(TabSplit)
        If Len(TabSplit(compteur)) = 0 Then Exit For
    Next compteur
    If compteur <= UBound(TabSplit) Then 'sorti avec Exit For
        TestSortie = True
    End If
End Function
```

Ce genre de test demeure cependant assez rare puisqu'il est souvent plus pratique de gérer le cas dans le test contenant la clause de sortie, ce qui dans l'exemple précédent s'écrirait :

```
Private Function TestSortie(ByVal Phrase As String) As Boolean

Dim compteur As Long, TabSplit() As String

    TabSplit = Split(Phrase, " ")
    For compteur = 0 To UBound(TabSplit)
        If Len(TabSplit(compteur)) = 0 Then
            TestSortie = True
            Exit For
        End If
    Next compteur

End Function
```

Do…Loop

Les boucles Do…Loop, parfois appelées boucles conditionnelles utilisent un test pour vérifier s'il convient de continuer les itérations. VBA utilise deux tests différents While (Tant que) qui poursuit les itérations tant que l'expression du test est vrai ou Until (Jusqu'à) qui poursuit les itérations jusqu'à que l'expression du test soit vraie.

La position du test va influer sur le nombre d'itérations. Si le test est positionné en début de boucle, c'est-à-dire sur la même ligne que le Do, il n'y aura par forcément exécution du traitement contenu dans la boucle, par contre s'il est positionné à la fin, c'est-à-dire sur la même ligne que Loop, le traitement sera toujours exécuté au moins une fois.

Quelle que soit la construction choisie, il est toujours possible d'utiliser la clause de sortie '*Exit Do*'.

Nous avons donc quatre formes de boucles conditionnelles qui sont :

Do While *Expression* Traitements [**Exit Do**] [Traitements] **Loop**	**Do** Traitements [**Exit Do**] [Traitements] **Loop While** *Expression*	**Do Until** *Expression* Traitements [**Exit Do**] [Traitements] **Loop**	**Do** Traitements [**Exit Do**] [Traitements] **Loop Until** *Expression*

Vous trouverez parfois une ancienne syntaxe équivalente à la première forme, la boucle While…Wend. Sa forme est :

While *Expression*
Traitements
Wend

Bien qu'équivalente fonctionnellement à la boucle Do While…Loop, on n'utilise plus tellement cette syntaxe car elle ne permet pas l'utilisation d'une clause de sortie ni le positionnement du test en fin de boucle.

Le choix de la boucle dépend généralement du contexte du code comme nous allons le voir dans quelques exemples simples.

```
Private Function ConvArabeVersRomain(ByVal Valeur As Integer) As String

    Do While Valeur > 0
        Select Case Valeur
            Case Is >= 1000
                ConvArabeVersRomain = ConvArabeVersRomain & "M"
                Valeur = Valeur - 1000

            Case Is >= 900
                ConvArabeVersRomain = ConvArabeVersRomain & "CM"
                Valeur = Valeur - 900

            Case Is >= 500
                ConvArabeVersRomain = ConvArabeVersRomain & "D"
                Valeur = Valeur - 500

            Case Is >= 400
                ConvArabeVersRomain = ConvArabeVersRomain & "CD"
                Valeur = Valeur - 400

            Case Is >= 100
                ConvArabeVersRomain = ConvArabeVersRomain & "C"
                Valeur = Valeur - 100

            Case Is >= 90
                ConvArabeVersRomain = ConvArabeVersRomain & "XC"
                Valeur = Valeur - 90

            Case Is >= 50
                ConvArabeVersRomain = ConvArabeVersRomain & "L"
```

```
                Valeur = Valeur - 50

            Case Is >= 40
                ConvArabeVersRomain = ConvArabeVersRomain & "XL"
                Valeur = Valeur - 40

            Case Is >= 10
                ConvArabeVersRomain = ConvArabeVersRomain & "X"
                Valeur = Valeur - 10

            Case 9
                ConvArabeVersRomain = ConvArabeVersRomain & "IX"
                Valeur = Valeur - 9

            Case Is >= 5
                ConvArabeVersRomain = ConvArabeVersRomain & "V"
                Valeur = Valeur - 5

            Case 4
                ConvArabeVersRomain = ConvArabeVersRomain & "IV"
                Valeur = Valeur - 4

            Case Else
                ConvArabeVersRomain = ConvArabeVersRomain & "I"
                Valeur = Valeur - 1
        End Select
    Loop
End Function
```

Dans ce cas, le test se met en début de boucle puisque l'exécution d'une itération ne doit avoir lieu que si la valeur testée est supérieure à 0.

La nature du test à choisir est souvent plus subtile. De fait, nous pourrions écrire :

```
    Do Until Valeur = 0
```

Et obtenir un comportement tout à fait similaire. Cependant, la syntaxe While nous permet d'éliminer aussi les valeurs négatives alors que l'utilisation de Until ne le permet pas.

L'exemple suivant montre l'utilisation d'un test Until en fin de boucle.

```
Private Function ChercherCellules(PlageCible As Range, ValeurCherchee As
Variant) As Range

    Dim CelluleTrouvee As Range, Adresse As String
    Set CelluleTrouvee = PlageCible.Find(ValeurCherchee)
    If Not CelluleTrouvee Is Nothing Then
        Adresse = CelluleTrouvee.Address
        Do
            If ChercherCellules Is Nothing Then Set ChercherCellules =
CelluleTrouvee Else Set ChercherCellules =
Application.Union(ChercherCellules, CelluleTrouvee)
            Set CelluleTrouvee = PlageCible.FindNext(CelluleTrouvee)
        Loop Until CelluleTrouvee.Address = Adresse
    End If

End Function
```

Énumérations & collections

Un objet Collection est un jeu d'éléments indexés auxquels il peut être fait référence comme s'ils constituaient un ensemble unique. Chaque élément d'une collection (appelés membre) peut être retrouvé à l'aide de son index. Les membres n'ont pas obligatoirement le même type de données, bien que ce soit presque toujours le cas dans les collections du modèle objet Excel.

Une collection présente toujours une propriété *Count* qui renvoie le nombre d'éléments, deux méthodes *Add* et *Remove* qui permettent d'ajouter ou de supprimer un élément, et une méthode *Item* qui permet de renvoyer un élément en fonction de son Index. L'index peut parfois être indifféremment un numéro d'ordre et/ou un nom.

Il existe une boucle particulière qui permet de parcourir tous les éléments d'une collection, la boucle For Each…Next, appelée aussi énumération. De la forme

For Each *Elément* **In** Collection
    Traitement
    [Exit For]
    [Traitement]
Next [*Elément*]

```
Public Sub DemoCollection()

    Dim MaColl As New Collection, Enumerateur As Variant

    MaColl.Add 1
    MaColl.Add "Démo"
    MaColl.Add #10/28/2005#
    MaColl.Add True
    For Each Enumerateur In MaColl
        MsgBox Enumerateur
        If IsDate(Enumerateur) Then Exit For
    Next Enumerateur

End Sub
```

Le modèle objet Excel expose un grand nombre de collection, nous les reverrons donc bientôt plus en détail.

Fonctions VBA

Pour finir l'étude du langage à proprement parler et avant de se lancer dans l'étude du modèle objet Excel, nous allons parcourir les fonctions dites intrinsèques du langage. Ces fonctions appartiennent à l'objet VBA et font souvent partie du scope, elles n'ont donc généralement pas besoin d'être qualifiées[3]. Je ne rentrerais pas ici dans l'architecture de l'objet VBA car cela n'a guère d'intérêt pour la programmation d'Excel, mais sachez juste que ces fonctions s'utilisent comme les fonctions ou procédures que vous écrivez vous-même. La présentation des fonctions qui va suivre ne sera pas exhaustive ni pour les fonctions, ni pour les arguments qu'elles acceptent. En effet, certaines fonctions utilisent de nombreux paramètres optionnels (que vous pouvez passer par position ou en les nommant) et la liste qui va suivre n'a pas pour but de doublonner l'aide en ligne, mais de vous montrer les fonctions les plus utilisées avec des exemples 'classiques'. Dans le cas où les détails complets sur la fonction sont nécessaires, il suffit de taper le nom de la fonction dans l'éditeur de code, de positionner le curseur sur ce nom et d'appuyer sur F1 pour afficher l'aide correspondante.

Dans les modèles donnés, les arguments entre crochets [] sont optionnels ; la notation […] signifie qu'il existe d'autres arguments optionnels qui ne sont pas utilisés dans ce cours.

Fonctions de conversions

Conversion de type

CBool(*expression*)	➢ **Boolean**	Les valeurs numériques sont convertit selon la règle toutes valeurs différentes de zéro est vrai.
CByte(*expression*)	➢ **Byte**	
CCur(*expression*)	➢ **Currency**	
CDate(*expression*)	➢ **Date**	Convertit les littéraux en date, et les nombres en suivant la règle "Partie entière égale au nombre de jours depuis le 1/1/1900 ; partie décimale égale au nombre de seconde depuis minuit.
CDbl(*expression*)	➢ **Double**	
CDec(*expression*)	➢ **Decimal**	
CInt(*expression*)	➢ **Integer**	Arrondit les décimaux au 0.5 près
CLng(*expression*)	➢ **Long**	Arrondit les décimaux au 0.5 près
CSng(*expression*)	➢ **Single**	
CStr(*expression*)	➢ **String**	N'échoue que pour les valeurs particulières Nothing, Empty, Error
CVar(*expression*)	➢ **Variant**	N'échoue jamais

Nous avons vu rapidement les fonctions de conversion de type lors de l'étude des types de données. Ces fonctions attendent une variable ou une expression d'un type et le transforme dans un autre type. Si la conversion échoue une erreur récupérable est levée.

[3] Il existe un cas dans Excel 2000 ou le compilateur détecte une ambiguïté entre la fonction Left et la propriété Left des UserForms. Il suffit d'écrire VBA.Left pour désigner la fonction et lever l'ambiguïté.

Conversions spécifiques

CVErr

CVErr(*NumErreur As integer*) **As Variant**

Permet de renvoyer un Variant contenant l'erreur spécifiée comme argument. On utilise généralement cette fonction dans deux cas.

Pour différer le traitement d'une erreur ou pour permettre le traitement par l'appelant, comme dans l'exemple suivant

```
Public Sub Traitement()

    Dim FichierLog As String, MaCell As Range, Reponse As Variant

    FichierLog = "d:\user\liste1.txt"
    For Each MaCell In ThisWorkbook.Worksheets(1).UsedRange.Cells
        If Not MaCell.Comment Is Nothing Then
            Reponse = AjouteAuFichier(FichierLog, MaCell.Comment.Text)
            If IsError(Reponse) Then
                Select Case CInt(Reponse)
                    Case 53
                        MsgBox "Fichier introuvable"

                    Case 55
                        MsgBox "Fichier déjà ouvert"

                End Select
            End If
        End If
    Next MaCell

End Sub

Private Function AjouteAuFichier(ByVal Chemin As String, ByVal Ajout As
String) As Variant

    On Error GoTo Erreur
    Open Chemin For Append As #1
    Print #1, Ajout
    Close #1
    AjouteAuFichier = True
    Exit Function
Erreur:
    AjouteAuFichier = CVErr(Err.Number)
    Err.Clear
End Function
```

On l'utilise aussi pour renvoyer des erreurs dans la cellule de calcul lorsqu'on écrit une fonction personnalisée de feuille de calcul.

```
Public Function ConvPSI2Pascal(ByVal Cellule As Range) As Variant

    If Not IsNumeric(Cellule.Value) Or IsEmpty(Cellule.Value) Then
        ConvPSI2Pascal = CVErr(xlErrNum)
    Else
        ConvPSI2Pascal = 6894.757 * Cellule.Value
    End If

End Function
```

Val

Val(*string As String*) **As type numérique**

La fonction Val cherche à interpréter une chaîne comme une valeur numérique. Elle lit les caractères de la gauche vers la droite et arrête l'interprétation dès qu'un caractère ne peut plus être interprété comme une partie d'un nombre.

```
 Public Sub TestVal()

Dim Nombre As Double
    Nombre = Val("121")
    Debug.Print Nombre 'renvoie 121
    Nombre = Val("121.10")
    Debug.Print Nombre 'renvoie 121.1
    Nombre = Val("121.10erp")
    Debug.Print Nombre 'renvoie 121.1
    Nombre = Val("a121")
    Debug.Print Nombre 'renvoie 0

End Sub
```

Format, Format$

Format(*expression As Variant* [, *format*] as String, […]) **As Variant**

Format$(*expression As Variant* [, *format*] as String, […]) **As String**

La fonction format cherche à transformer une valeur ou une expression en chaîne formatée selon l'argument de mise en forme.

Cette fonction accepte comme argument format soit des éléments prédéfinis, soit la construction de spécification de formatage personnalisée.

Les chaînes prédéfinies utilisables sont

Argument format	Description
Standard	Affichage de la date et/ou de l'heure. Pour des nombres réels, affichage de la date et de l'heure, par exemple 4/3/93 05:34 PM. S'il n'y a pas de partie décimale, affichage de la date seulement, par exemple 4/3/93. S'il n'y a pas de partie entière, affichage de l'heure seulement, par exemple, 05:34 PM. Le format de la date est déterminé par les paramètres de votre système.
Long Date	Affichage de la date complète selon le format défini dans votre système.
Medium Date	Affichage de la date selon le format intermédiaire conforme à la langue de l'application hôte.
Short Date	Affichage de la date abrégée selon le format défini dans votre système.
Long Time	Affichage de l'heure complète selon le format défini dans votre système, comprenant les heures, les minutes et les secondes.
Medium Time	Affichage de l'heure dans un format de 12 heures en utilisant les heures et les minutes ainsi que les indicateurs AM/PM.
Short Time	Affichage de l'heure au format de 24 heures, par exemple 17:45.
General Number	Affichage du nombre sans séparateur de milliers.
Currency	Affichage du nombre avec un séparateur de milliers, le cas échéant ; affichage de deux chiffres à droite du séparateur décimal. Le résultat est fonction des paramètres régionaux de votre système.

Argument format	Description
Fixed	Affichage d'au moins un chiffre à gauche et de deux chiffres à droite du séparateur décimal.
Standard	Affichage d'un nombre avec séparateur de milliers et d'au moins un chiffre à gauche et de deux chiffres à droite du séparateur décimal.
Percent	Affichage d'un nombre multiplié par 100 suivi du signe pourcentage (**%**) ; affichage automatique de deux chiffres à droite du séparateur décimal.
Scientific	Utilisation de la notation scientifique standard.
Yes/No	Affichage de Non si le nombre est 0 ; sinon affichage de Oui.
True/False	Affichage de **Faux** si le nombre est 0 ; sinon affichage de **Vrai**.
On/Off	Affichage de Inactif si le nombre est 0 ; sinon affichage de Actif.

Les codes personnalisés étant assez nombreux, je n'en reprendrais pas la liste ici. Vous trouverez ceux-ci dans l'aide en ligne.

Quelques exemples :

```
Public Sub TestFormat()

Dim Chaîne As String, Nombre As Double, UneDate As Date

    Chaîne = "Avec Casse"
    Nombre = "1253.124"
    UneDate = #1/16/1967 8:15:00 PM#

    Debug.Print Format(Chaîne, "<") 'avec casse
    Debug.Print Format(Chaîne, ">") 'AVEC CASSE
    Debug.Print Format(Chaîne, "@@@@@!") 'Casse

    Debug.Print Format(Nombre, "Standard") '1 253.12
    Debug.Print Format(Nombre, "Percent") '125312.40%
    Debug.Print Format(Nombre, "Scientific") '1.25E+03
    Debug.Print Format(Nombre, "0.00") '1253.12

    Debug.Print Format(UneDate, "Long Date") 'lundi 16 janvier 1967
    Debug.Print Format(UneDate, "Short Date") '16/01/1967
    Debug.Print Format(UneDate, "Long Time") '20:15:00
    Debug.Print Format(UneDate, "ddd dd/mm/yyyy") 'lun. 16/01/1967
    Debug.Print Format(UneDate, "ww") '3
    Debug.Print Format(UneDate, "h:mm") '20:15

End Sub
```

Conversion de valeur

Hex, Hex$

Hex(*number As Variant*) **As Variant**
Hex$(*number As Variant*) **As String**

Convertit un nombre décimal en Hexadécimal. L'argument 'number' doit renvoyer un nombre entier sinon le résultat sera arrondi à l'entier le plus proche. Notez que :

Hex(Null) = Null

Hex(Empty)=0

Les nombres hexadécimaux sont parfois utilisés dans le code, préfixés par &H. Par exemple &H10 vaut 16.

Oct, Oct$

Oct(*number As Variant*) **As Variant**
Oct$(*number As Variant*) **As String**

Convertit un nombre décimal en Octal. L'argument 'number' doit renvoyer un nombre entier sinon le résultat sera arrondi à l'entier le plus proche. Notez que :

Oct(Null) = Null

Oct(Empty)=0

Les nombres octaux sont parfois utilisés dans le code, préfixés par &O. Par exemple &O10 vaut 8.

Int, Fix

Int(*number As Variant*) **As Variant**
Fix(*number As Variant*) **As Variant**

Renvoie la partie entière de l'argument. Pour les valeurs positives, **Int** et **Fix** renvoient la même valeur, pour les valeurs négatives, **Int** renvoie le premier entier négatif inférieur ou égal à *number*, alors que **Fix** renvoie le premier entier négatif supérieur ou égal à *number*. Autrement dit, **Int**(*-7.3*) vaut -8 alors que **Fix**(*-7.3*) vaut -7.

Fonctions de Date & Heure

La manipulation des dates et des heures est souvent problématique du fait de la méthode de stockage des dates et des différences de format de dates entre le système anglo-saxon et le notre.

Pour s'affranchir de cet aspect international, Excel stocke les dates sous formes d'un nombre décimal composite définit comme :

Une partie entière représentant le nombre de jours écoulés depuis le 01/01/1900 celui-ci étant compté.

Une partie décimale représentant le nombre de secondes depuis 00 h 00 min 00 s, ramenée à 1 jour (c'est à dire le temps en seconde / 86400 s de la journée)

Cette méthode de stockage permet de limiter la taille de stockage des dates et de pouvoir utiliser l'arithmétique décimale sur les dates. Cette représentation numérique est souvent appelée "numéro de série" ou la partie entière est le numéro de série de la date et la partie décimale celui de l'heure.

Toutes les représentations du temps étant sous forme de date, il n'existe pas de représentation de durée sous forme de temps supérieure à 23:59:59, hors artifice de formatage. Ainsi vous pouvez afficher la valeur 24:00:00 dans une cellule Excel en appliquant le format [h]:mm:ss, mais vous verrez que la valeur réelle de la cellule sera 02/01/1900 00:00:00.

Une fois que vous avez à l'esprit ce mode de stockage, la manipulation des dates ne pose pas plus de difficulté qu'autre chose.

Prenons l'exemple suivant :

```
Sub Test()

Dim Date1 As Date, EDate As Date, ETemps As Date, TempSec As Double
Dim Heure As Long, Minute As Long, Seconde As Long

    Date1 = #10/22/2002 2:02:21 PM#
    EDate = CLng(Date1)
    ETemps = Date1 - Int(Date1)
    Debug.Print "Date : " & EDate & " -> N° série : " & CLng(EDate) &
vbNewLine
    'Date : 23/10/2002 -> N° série : 37552
    Debug.Print "Heure : " & ETemps & " -> N° série : " & CDbl(ETemps)
    'Heure : 14:02:21 -> N° série : 0.5849653
    TempSec = Round(CDbl(ETemps) * 86400)
    Heure = TempSec \ 3600
    Minute = (TempSec - Heure * 3600) \ 60
    Seconde = TempSec - (Heure * 3600) - Minute * 60
    Debug.Print Heure & ":" & Minute & ":" & Seconde
    '14:2:21

End Sub
```

Notons toutefois que ce code fonctionne parce qu'on utilise un arrondi pour retrouver le temps en seconde et non une troncature.

Récupération du temps système

Date, Date$

Renvoie la date système, Date renvoie un variant alors que Date$ renvoie une chaîne de caractères. La chaîne renvoyée par Date$ est au format international.

Time, Time$

Renvoie l'heure système, Time renvoie un Variant alors que Time$ renvoie une chaîne de caractères.

Timer

Renvoie une valeur de type **Single** représentant le nombre de secondes écoulées depuis minuit.
Notez que Round(Time) renvoie la même valeur que Int(Timer).

Now

Renvoie la date et l'heure système (Variant). Beaucoup de développeurs utilisent cette fonction dans tous les cas pour récupérer le temps système pour éviter la confusion entre les fonctions Date, Time, et les instructions du même nom.

Fonctions de conversions

DateValue, TimeValue

On utilise généralement la fonction de conversion CDate pour convertir une chaîne ou une expression en date. Cependant lorsque la valeur contient une information de date et de temps et qu'on ne souhaite récupérer qu'une des deux informations, il peut être plus efficace d'utiliser les fonctions DateValue et/ou TimeValue.

```
Sub Test()

    Dim sDate As String
    sDate = "10 décembre 2002 2:14:17 PM"
    Debug.Print CDate(sDate)
    '10/12/2002 14:14:17
    Debug.Print DateValue(sDate)
    '10/12/2002
    Debug.Print TimeValue(sDate)
    '14:14:17

End Sub
```

DateSerial

DateSerial(*year As Integer*, *month As Integer*, *day As Integer*) **As Variant**

Renvoie le numéro de série de la date correspondant aux arguments passés.
Par exemple :

```
Debug.Print DateSerial(2002, 10, 22)
'22/10/2002
```

Attention, il n'y a pas de contrôles partiels sur les arguments passés, seul l'appartenance à la plage des valeurs autorisées pour les dates est contrôlée.

Autrement dit, on peut saisir des valeurs supérieures à 31 pour les jours et supérieures à 12 pour les mois, voire des valeurs négatives.

```
Sub Test()

    Debug.Print DateSerial(2002, 10, 22)
    '22/10/2002
    Debug.Print DateSerial(2002, 10, 33)
    '02/11/2002
    Debug.Print DateSerial(2002, 11, 33)
    '03/12/2002
    Debug.Print DateSerial(2002, 14, 33)
    '05/03/2003
    Debug.Print DateSerial(2002, 15, 33)
    '02/04/2003
    Debug.Print DateSerial(2002, 36, 33)
    '02/01/2005
    Debug.Print DateSerial(2002, -36, 33)
    '02/01/1999

End Sub
```

Nous reviendrons sur le fonctionnement de cette fonction lors de l'étude de la fonction DateAdd

N.B : Vous noterez que la demande d'aff hage de la fonction DateSerial par Debug.Print renvoie une date et non un numéro de série. Ceci provient du fait que la valeur renvoyé est un variant d'abord interprété comme une date. Vous devez le récupérer dans une variable de type Long pour récupérer le numéro de série.

TimeSerial

TimeSerial(*Hour As Integer*, *minute As Integer*, *second As Integer*) **As Variant**

Renvoie le numéro de série de l'heure correspondant aux arguments passés. La encore, il n'y a pas de contrôles partiels des arguments mais juste l'obligation que le résultat soit dans la plage des dates valides.

```
Sub Test()

    Debug.Print TimeSerial(12, 10, 52)
    '12:10:52
    Debug.Print TimeSerial(12, -10, 52)
    '11:50:52
    Debug.Print TimeSerial(18, 90, 81)
    '19:31:21
    Debug.Print TimeSerial(22, 120, 52)
    '31/12/1899 00:00:52

End Sub
```

N.B : Même rem nction DateSerial

Fonctions d'extraction

Fonctions spécifiques

Les fonctions spécifiques renvoient un élément de la date sous forme d'un Integer.

NomFonction(*date As Date*) **As Integer**

Nom de la fonction	*Résul*
Day	1 < R < 31
Month	1 < R < 12
Year	100 < R < 9999
Hour	0 < R < 23
Minute	0 < R < 59
Seconde	0 < R < 59

WeekDay

La fonction WeekDay renvoie un numéro d'ordre correspondant à la position du jour dans la semaine par rapport à la base donnée en argument.

Weekday(*date As Date,* [*firstdayofweek*]) **As Integer**

La constante 'firstdayofweek' donne la base pour le décompte. Si c'est un jour spécifique, cela veut dire que ce jour aura le numéro de série 1, si c'est la base système, cela dépendra des paramètres internationaux (normalement le lundi en France), si le paramètre est omis la base sera le dimanche

Vous trouverez le tableau des constantes dans l'étude de la fonction suivante.

```
Sub Test()
    Dim MaDate As Date
    MaDate = #10/22/2002 5:14:26 PM#
    Debug.Print Day(MaDate) '22
    Debug.Print Month(MaDate) '10
    Debug.Print Year(MaDate) '2002
    Debug.Print Hour(MaDate) '17
    Debug.Print Minute(MaDate) '14
    Debug.Print Second(MaDate) '26
    Debug.Print Weekday(MaDate, vbSunday) '3
    Debug.Print Weekday(MaDate, vbMonday) '2
    Debug.Print Weekday(MaDate, vbUseSystemDayOfWeek) '2
End Sub
```

DatePart

Cette fonction renvoie des informations sur la date en fonction des arguments passés.

DatePart(*interval As Stringl, date As Date*[*,firstdayofweek, firstweekofyear*]]) **As Integer**

L'argument *interval* est une chaîne prédéfinie définissant le type d'information que l'on souhaite récupérer. Elle peut prendre les valeurs :

Valeur	Information	
yyyy	Année	100 ≤ x ≤ 9999
q	Trimestre	1 ≤ x ≤ 4
m	Mois	1 ≤ x ≤ 12
y	Jour de l'année	1 ≤ x ≤366
d	Jour	1 ≤ x ≤31
w	Jour de la semaine	1 ≤ x ≤ 7 : Sensible à l'argument *firstdayofweek*
ww	Semaine	1 ≤ x ≤ 53 : Sensible à l'argument *firstweekofyear*
h	Heure	0 ≤ x ≤ 23
n	Minute	0 ≤ x ≤ 59
s	Seconde	0 ≤ x ≤ 59

L'argument *firstdayofweek* définit quel jour de la semaine aura le numéro de série 1. Il n'est pas nécessaire de le préciser pour récupérer d'autres informations, sauf dans certaines combinaisons avec l'argument *firstweekofyear* pour déterminer le numéro de semaine. Lorsqu'il est omis, c'est le dimanche qui sera considéré comme le premier jour de la semaine. Il peut prendre comme valeurs une des constantes suivantes :

Constante	Valeur	Description
vbUseSystem	0	Définit par les paramètres internationaux système
vbSunday	1	Dimanche (valeur par défaut)
vbMonday	2	Lundi
vbTuesday	3	Mardi
vbWednesday	4	Mercredi
vbThursday	5	Jeudi
vbFriday	6	Vendredi
vbSaturday	7	Samedi

L'argument *firstweekofyear* précise le mode de définition de la semaine de l'année ayant le numéro de série 1. Il n'est utile de le préciser que lorsqu'on veut extraire le numéro de série de la semaine. Lorsqu'il est omis, ce paramètre prend la semaine du 1er Janvier comme semaine numéro 1. En France c'est la première semaine de 4 jours qui a le numéro de semaine. Il peut prendre comme valeur une des constantes suivantes :

Constante	Valeur	Description
vbUseSystem	0	Définit par les paramètres internationaux système
vbFirstJan1	1	Semaine du 1er janvier (valeur par défaut).
vbFirstFourDays	2	Première semaine comportant au moins quatre jours dans l'année nouvelle.
vbFirstFullWeek	3	Première semaine complète de l'année.

Quelques exemples d'utilisation :

```
Sub Test()

    Dim MaDate As Date
    MaDate = #10/22/2002 5:14:26 PM#

    Debug.Print DatePart("yyyy", MaDate) '2002
    Debug.Print DatePart("q", MaDate) '4
    Debug.Print DatePart("m", MaDate) '10
    Debug.Print DatePart("y", MaDate) '295
    Debug.Print DatePart("d", MaDate) ' 22
    Debug.Print DatePart("w", MaDate) '3
    Debug.Print DatePart("w", MaDate, vbMonday) '2
    Debug.Print DatePart("ww", MaDate) '43
    Debug.Print DatePart("ww", MaDate, vbMonday, vbFirstFullWeek) '42
    Debug.Print DatePart("h", MaDate) '17
    Debug.Print DatePart("n", MaDate) '14
    Debug.Print DatePart("s", MaDate) '26

End Sub
```

Fonctions de calculs

DateAdd

Permet d'ajouter ou de soustraire un intervalle de temps sur une date. De la forme :

DateAdd(*interval* ***As String,*** *number* ***As Long,*** *date* ***As Date***)

L'argument *interval* est une chaîne prédéfinie déterminant l'intervalle de temps à ajouter, cette chaîne peut prendre les mêmes valeurs que celles de la fonction DatePart.

La valeur *number* détermine le nombre d'intervalles. Une valeur décimale sera arrondie à l'entier le plus proche. Une valeur négative engendrera une soustraction.

Quelques exemples d'utilisation :

```
Sub Test()
    Dim MaDate As Date
    MaDate = #1/31/2002 5:14:26 PM#
    Debug.Print DateAdd("s", 40, MaDate)
    '31/01/2002 17:15:06
    Debug.Print DateAdd("n", -15, MaDate)
    '31/01/2002 16:59:26
    Debug.Print DateAdd("w", 5, MaDate)
    '05/02/2002 17:14:26
    Debug.Print DateAdd("y", -1992, MaDate)
    '18/08/1996 17:14:26
    Debug.Print DateAdd("m", 25, MaDate)
    '29/02/2004 17:14:26
    Debug.Print DateAdd("h", 40, MaDate)
    '02/02/2002 09:14:26
    Debug.Print DateAdd("yyyy", -1992, MaDate)
    'Erreur hors de la plage des dates
End Sub
```

Comme vous le voyez, la fonction est assez simple d'emploi. Quelques remarques sont toutefois nécessaires :

- Il n'est pas possible de travailler sur un nombre décimal d'intervalles. Pour obtenir une modification équivalente vous pouvez soit imbriquer les fonctions DateAdd, soit les utiliser en succession.
- L'emploi des littéraux "d", "w", "y" est équivalent pour l'ajout ou la soustraction de jour.
- Lorsque l'intervalle est défini par "m", "yyyy", "q", il ne peut pas avoir pour conséquence d'engendrer la modification d'une autre partie de la date. Autrement dit, si vous ajoutez des mois ou des trimestres à une date dont le jour est 31 et que le résultat conduit à un mois qui n'a que 30 jours (ou 28) c'est le dernier jour du mois qui sera renvoyé. Il en sera de même pour les années bissextiles avec le littéral "yyyy".

```
Sub Test()

    Dim MaDate As Date
    MaDate = #1/31/2000 5:14:26 PM#
    Debug.Print DateAdd("n", 30, DateAdd("h", 2, MaDate))
    '29/02/2000 19:44:26
    MaDate = DateAdd("h", 2, MaDate)
    Debug.Print DateAdd("n", 30, MaDate)
    '29/02/2000 19:44:26
    Debug.Print DateAdd("yyyy", 1, MaDate)
    '28/02/2001 19:14:26
    Debug.Print DateAdd("m", 12, MaDate)
    '28/02/2001 19:14:26
    Debug.Print DateAdd("q", 1, MaDate)

End Sub
```

Une bonne pratique de cette fonction permet de mieux comprendre le fonctionnement de la fonction DateSerial que nous avons vu précédemment. Nous allons écrire notre propre fonction DateSerial pour mieux le voir.

Cet exemple va nous emmener un petit peu loin, si vous débutez le VBA vous pouvez continuer à l'étude de la fonction suivante.

Notre fonction sera :

```
Public Function NotreDateSerial(ByVal Annee As Integer, ByVal Mois As
Integer, ByVal Jour As Integer) As Variant

    Dim BaseDate As Date
    BaseDate = #1/31/1999#
    'Test sur l'année
    Select Case Annee
        Case Is > 100 'affectation de l'année
        BaseDate = DateAdd("yyyy", -1 * (1999 - Annee), BaseDate)

        Case Is <= 0 'soustraire à la base
            BaseDate = DateAdd("yyyy", Annee, BaseDate)

        Case Else 'lire comme Base + Annee - 100
            BaseDate = DateAdd("yyyy", Annee - 100, BaseDate)

    End Select
    'aller au dernier jour de novembre et ajouter Mois
    BaseDate = DateAdd("m", 10 + Mois, BaseDate)
    'si l'année est une affectation, enlever un an
    If Annee > 100 Then BaseDate = DateAdd("yyyy", -1, BaseDate)
    'ajouter les jours
    BaseDate = DateAdd("d", Jour, BaseDate)
    NotreDateSerial = BaseDate

End Function
```

Pourquoi un code aussi complexe. Comme nous l'avons dit au cours de l'étude de la fonction DateSerial, il n'y a pas de contrôle unitaire des arguments passés. Contrairement à une idée reçue, DateSerial ne déclenche pas d'erreur si on lui passe une valeur d'année inférieure à 100. En fait, DateSerial gère une date en interne, défini au dernier jour de novembre 1999. Je ne dis pas au 30 Novembre car DateSerial renverra le 31 décembre si vous ajoutez un mois à la date interne.

Pour pouvoir gérer cette condition, je passe donc par une date de base ayant 31 jours ce qui utilisera les propriétés de l'ajout avec l'argument "m", tel que nous l'avons vu précédemment.

DateDiff

Renvoie la différence entre deux dates dans l'intervalle spécifié. De la forme :

DateDiff(*interval* ***As String,*** *date1* ***As Date,*** *date2* ***As Date***[**,** *firstdayofweek*[**,** *firstweekofyear*]]) **As Long**

Où l'argument *interval* est une chaîne prédéfinie précisant l'unité de temps de la valeur retournée. Elle peut prendre les mêmes valeurs que pour la fonction DatePart.

Cette fonction quoique très pratique est un peu piégeuse car la notion d'intervalle de temps va dépendre de paramètres plus numériques que logiques.

Regardons le code suivant :

```
Sub Test()

    Debug.Print DateDiff("d", DateValue("27/02/2002"),
DateValue("31/12/2002")) '307
    Debug.Print DateDiff("yyyy", DateValue("27/02/2002"),
DateValue("31/12/2002")) '0
    Debug.Print DateDiff("d", DateValue("31/12/2002"),
DateValue("15/01/2003")) '15
    Debug.Print DateDiff("yyyy", DateValue("31/12/2002"),
DateValue("15/01/2003")) '1

    Debug.Print DateDiff("d", DateValue("02/02/2002"),
DateValue("28/02/2002")) '26
    Debug.Print DateDiff("m", DateValue("02/02/2002"),
DateValue("28/02/2002")) '0
    Debug.Print DateDiff("d", DateValue("28/02/2002"),
DateValue("01/03/2002")) '1
    Debug.Print DateDiff("m", DateValue("28/02/2002"),
DateValue("01/03/2002")) '1

End Sub
```

Comme vous le voyez, une différence de 300 jours renverra une différence de 0 années alors qu'une différence de 15 jours peut renvoyer un intervalle de 1 année. Il en est de même avec les mois.

De fait le test se fait sur la valeur des éléments composant la date comparés un à un. Vous devez donc utiliser la fonction DateDiff en ayant bien à l'esprit le mode de calcul utilisé. Pour éviter les erreurs, on utilise généralement un retour dans la plus petite unité de l'élément (jours pour la date, seconde pour le temps) afin d'avoir la durée la plus juste possible.

Si Date1 est postérieure à Date2, la fonction DateDiff renvoie un nombre négatif.

Exemples classiques d'utilisation

```
Public Function FinDuMois(ByVal D As Date) As Date
    FinDuMois = DateSerial(Year(D), Month(D) + 1, 0)
End Function

Public Function DebutSemaine(ByVal D As Date) As Date
    DebutSemaine = D - Weekday(D) + 7
End Function

Public Function FinSemaine(ByVal D As Date) As Date
    FinSemaine = D - Weekday(D) + 7
End Function

Public Function EstBissextile(ByVal Annee As Long) As Boolean
    EstBissextile = Annee Mod 4 = 0 And (Annee Mod 100 <> 0 Or Annee Mod
400 = 0)
End Function

Public Function NombreMemeJour(ByVal Date1 As Date, ByVal Date2 As Date,
Jour As VbDayOfWeek) As Integer
    NombreMemeJour = DateDiff("ww", Date1, Date2, Jour)
End Function

Public Function Duree(ByVal Date1 As Date, ByVal Date2 As Date) As Date
    Duree = DateDiff("s", Date1, Date2) / 86400
End Function
```

```vba
Public Function DimanchePaques(ByVal Annee As Integer) As Date
    'algorithme de Oudin
    Dim G As Integer, C As Integer, C_4 As Integer, E As Integer
    Dim H As Integer, K As Integer, P As Integer, Q As Integer
    Dim I As Integer, B As Integer, J1 As Integer, J2 As Integer
    Dim R As Integer

    G = Annee Mod 19
    C = Annee \ 100
    C_4 = C \ 4
    E = (8 * C + 13) \ 25
    H = (19 * G + C - C_4 - E + 15) Mod 30
    K = H \ 28
    P = 29 \ (H + 1)
    Q = (21 - G) \ 11
    I = (K * P * Q - 1) * K + H
    B = Annee \ 4 + Annee
    J1 = B + I + 2 + C_4 - C
    J2 = J1 Mod 7
    R = 28 + I - J2

    If R <= 31 Then
        DimanchePaques = DateValue(CStr(R) & "/3/" & CStr(Annee))
    Else
        DimanchePaques = DateValue(CStr(R - 31) & "/4/" & CStr(Annee))
    End If

End Function

Public Function JoursFeries(ByVal Annee As Integer) As Variant

    ReDim JoursFeries(1 To 11)
    JoursFeries(1) = DateSerial(Annee, 1, 1) '1er janvier
    JoursFeries(2) = DateAdd("d", 1, DimanchePaques) 'lundi de paques
    JoursFeries(3) = DateSerial(Annee, 5, 1) '1er Mai
    JoursFeries(4) = DateSerial(Annee, 5, 8) 'victoire 1945
    JoursFeries(5) = DateAdd("d", 39, DimanchePaques) 'jeudi ascension
    JoursFeries(6) = DateAdd("d", 50, DimanchePaques) 'lundi pentecote
    JoursFeries(7) = DateSerial(Annee, 7, 14) 'fête nationale
    JoursFeries(8) = DateSerial(Annee, 8, 15) 'assomption
    JoursFeries(9) = DateSerial(Annee, 11, 1) 'toussaint
    JoursFeries(10) = DateSerial(Annee, 11, 11) 'armistice 14-18
    JoursFeries(11) = DateSerial(Annee, 12, 25) 'Noel

End Function
```

Fonctions de fichiers

Les fonctions de manipulation de fichiers que nous allons voir sont incluses dans le VBA. Il ne s'agit pas ici des méthodes de manipulation de la bibliothèque d'objet Excel, ni de celle de la librairie Office. Je ne parlerais pas non plus dans ce cours de la manipulation de la librairie FileSystemObject. Le terme fonctions est d'ailleurs un peu usurpé puisqu'en l'occurrence on utilisera aussi des instructions.

Les opérations sur les fichiers se divisent en deux, celles sur le système de fichiers et celles sur les fichiers.

Système de fichier

ChDir

Change le répertoire courant.

ChDir *path* ***As String***

Le répertoire désigné par *Path* doit forcément exister. L'instruction **ChDir** change le répertoire par défaut mais pas le lecteur par défaut. Si le lecteur n'est pas spécifié dans *Path*, la commande s'applique au lecteur par défaut.

ChDrive

Change le lecteur courant

ChDrive *Drive* ***As String***

CurDir

Renvoie le chemin courant

CurDir[(*drive*)]

Si l'argument *drive* est omis, c'est le chemin par défaut du lecteur par défaut qui est renvoyé sinon c'est celui du lecteur spécifié.

Dir

Pour les vieux qui ont connu le doux temps de MS-DOS, cette fonction ne présentera pas de difficultés de manipulation.

Dir[(*pathname*[**,** *attributes*])]

Pathname est un argument facultatif qui doit désigner un chemin existant, acceptant des caractères génériques. Selon sa nature, il permettra d'utiliser la fonction **Dir** différemment.

Attributes est un masque binaire acceptant les composants suivants :

Constante		Description
vbNormal	0	(Par défaut) Spécifie les fichiers sans attributs.
vbReadOnly	1	Spécifie les fichiers accessibles en lecture seule ainsi que les fichiers sans attributs.
vbHidden	2	Spécifie les fichiers cachés ainsi que les fichiers sans attributs.
vbSystem	4	Spécifie les fichiers système ainsi que les fichiers sans attributs. Non disponible sur le Macintosh.
vbVolume	8	Spécifie un nom de volume ; si un autre attribut est spécifié, la constante **vbVolume** est ignorée. Non disponible sur Macintosh.
vbDirectory	16	Spécifie les dossiers ainsi que les fichiers sans attributs.
vbArchive	32	Spécifie les fichiers modifiés ainsi que les fichiers sans attributs.

Dans l'absolu, Dir renvoie un nom de fichier ou de répertoire lorsqu'elle trouve une correspondance avec les arguments passés ou une chaîne vide dans le cas contraire. Lorsque plusieurs fichiers ou dossiers répondent aux arguments, Dir renvoie le premier élément de la liste. Un nouvel appel à la fonction Dir sans argument renverra l'élément suivant, et ainsi de suite jusqu'à l'obtention d'une chaîne vide. Les possibilités d'emplois sont nombreuses, aussi allons nous voir deux exemples classiques d'utilisation pour mieux comprendre son fonctionnement.

La première application permet de vérifier si un fichier existe sans utiliser le contrôle d'erreur.

```
Private Function FileExist(ByVal Chemin As String) As Boolean

    If Len(Dir(Chemin)) > 0 Then FileExist = True

End Function
```

La fonction se base sur le principe de la fonction Dir qui renvoie le nom de fichier passé en argument lorsque celui-ci existe. La même fonction peut se faire avec les répertoires.

```
Private Function FolderExist(ByVal Repertoire As String) As Boolean

    If Len(Dir(Repertoire, vbDirectory)) > 0 Then FolderExist = True

End Function
```

Le répertoire peut éventuellement se terminer par "\" ou non.

La deuxième fonction consiste à récupérer la liste des fichiers d'un répertoire en précisant éventuellement un masque de recherche. La fonction renvoie aussi le nombre de fichiers.

```
Private Function ListeFichier(ByVal Repertoire As String, ByRef Coll As
Collection, Optional ByVal Pattern As String = "*.*") As Integer

    If Right(Repertoire, 1) <> "\" Then Repertoire = Repertoire & "\"
    If Dir(Repertoire & "\") = "" Then Exit Function
    Dim MonFichier As String
    MonFichier = Dir(Repertoire & Pattern, vbArchive Or vbNormal Or
vbReadOnly)
    Do While Len(MonFichier) > 0
        Coll.Add (Repertoire & MonFichier)
        ListeFichier = ListeFichier + 1
        MonFichier = Dir
    Loop
End Function
```

Vous noterez que la liste des fichiers sera placée dans un argument collection passé par référence. Vous noterez également que le code teste l'absence d'un caractère antislash final et l'ajoute le cas échéant.

FileAttr

Cette fonction est souvent confondue avec la fonction GetAttr. La fonction FileAttr renvoie le mode d'ouverture d'un fichier ce qui n'est pas utile très souvent.

FileCopy

Copie le fichier passé comme argument source dans destination. Les deux arguments doivent comprendre le nom du fichier, mais pas nécessairement le même.

FileCopy *source As String, destination As String*

```
Private Sub Test1()

    FileCopy "d:\svg\essai.txt", "d:\svg\jmarc\essai.txt" 'valide
    FileCopy "d:\svg\essai.txt", "d:\svg\jmarc\nouveau.txt" 'valide
    FileCopy "d:\svg\essai.txt", "d:\svg\jmarc\" 'erroné

End Sub
```

FileDateTime

Renvoie la date et l'heure de dernière modification du fichier passé comme argument.
FileDateTime(*pathname As String*) **As Date**
Par exemple

```
Debug.Print FileDateTime("d:\svg\essai.txt")
'20/10/2002 21:35:13
```

FileLen

Renvoie la taille du fichier, en octets, passé en argument.
FileLen(*pathname As String*) **As Long**

```
Debug.Print FileLen("d:\svg\win32api.txt")
'667988
```

GetAttr & SetAttr

La fonction **SetAttr** définie les attributs du fichier passé comme premier argument à l'aide du masque binaire passé comme second argument. La fonction **GetAttr** renvoie un masque binaire des attributs du fichier passé en argument.
SetAttr *PathName As String, Attributes As VbFileAttribute*
GetAttr(*pathname As String*) **As VbFileAttribute**
Les attributs sont les mêmes que ceux donnés pour la fonction **Dir**.
L'exemple suivant va donner les attributs fichier caché et lecture seul à un fichier puis va relire ces attributs.

```
Private Sub TestAttribut()

Dim MesAttributs As VbFileAttribute, msg As String, ReadInt As Integer

    MesAttributs = MesAttributs Or vbHidden
    MesAttributs = MesAttributs Or vbReadOnly
    MesAttributs = MesAttributs Or GetAttr("d:\svg\essai.txt")
    SetAttr "d:\svg\essai.txt", MesAttributs
    MesAttributs = 0
    MesAttributs = GetAttr("d:\svg\essai.txt")
    Do While MesAttributs > 0
        ReadInt = Int(Log(MesAttributs) / Log(2))
        msg = msg & Choose(ReadInt + 1, "vbReadOnly", "vbHidden",
"vbSystem", "", "", "vbArchive", "vbAlias") & vbNewLine
        MesAttributs = MesAttributs - 2 ^ ReadInt
    Loop
    MsgBox msg
End Sub
```

Dans cet exemple, on obtient comme réponse :

Cela vient du fait que la troisième ligne de code a associé les attributs déjà existants aux attributs ajoutés par le code. Sans cette ligne, l'attribut archive aurait été effacé.

Kill

Supprime le ou les fichiers en fonction de l'argument.

Kill ***pathname As String***

Si *pathname* désigne un fichier, celui ci sera supprimé. Si, *pathname* contient des caractères génériques, tous les fichiers correspondant seront supprimés.

MkDir & RmDir

La fonction MkDir crée un répertoire dans le répertoire passé comme argument, ou dans le lecteur par défaut si l'argument est omis.

La fonction RmDir supprime le répertoire passé en argument. Celui-ci peut être un chemin relatif. Le répertoire cible ne doit pas contenir de fichier ni être le chemin courant.

MkDir ***pathname As String***

RmDir ***pathname As String***

Les deux procédures suivantes illustrent les manipulations du système de fichier.

```
Public Sub Creation()

Dim Fichier As String
Fichier = "d:\user\tutos\codemoteur.txt"
If Not FolderExist("d:\user\tutos\demo\") Then
    MkDir "d:\user\tutos\demo\"
    FileCopy Fichier, "d:\user\tutos\demo\code.txt"
    ChDrive "D"
    ChDir "d:\user\tutos\demo\"
End If

End Sub

Public Sub Suppression()
    If StrComp(CurDir, "d:\user\tutos\demo", vbTextCompare) = 0 Then
        Dim CollFichier As New Collection, NbFichier As Integer, strFichier
As Variant
        NbFichier = ListeFichier(CurDir, CollFichier, "*.*")
        If NbFichier > 0 Then
            For Each strFichier In CollFichier
                Kill strFichier
            Next
        End If
        ChDir "d:\user\tutos"
        RmDir "d:\user\tutos\demo"
    End If
End Sub
```

Le code de suppression pourrait être plus simple puisqu'il suffirait de faire :

Kill CurDir & "*.*"

N.B : Bien que ça n'ait rien à voir avec le sujet qui nous intéresse, vous noterez que l'énumérateur *strFichier* est déclaré comme Variant bien que nous sachions pertinemment que la collection ne contient que des chaînes. En VBA, l'énumérateur ne peut être que de type Variant ou Object.

Manipulation de fichier

Les fonctions standard de manipulation de fichiers permettent l'accès à trois types de fichiers :

- L'accès binaire ➢ Utilisé surtout pour réduire la taille des fichiers.
- L'accès aléatoire ➢ Utilisé pour une manipulation proche des SGBD.
- L'accès séquentiel ➢ Gère les fichiers texte.

Nous allons voir succinctement les différentes instructions et fonctions de manipulation de fichiers, puis un exemple de lecture écriture pour les trois types d'accès.

L'instruction Open

Pour pouvoir manipuler un fichier, il va forcément falloir commencer par l'ouvrir ou le créer.
L'instruction Open est de la forme :
Open *pathname* **For** *mode* [**Access** *access*] [*lock*] **As** [**#**]*filenumber* [**Len**=*reclength*]
Où

- *Pathname* désigne le fichier à ouvrir ou à créer si l'argument représente un fichier qui n'existe pas lors d'une opération d'écriture.
- *Mode* désigne le mode d'accès tel que :
  - Append ⇨ Ouvre un fichier texte (séquentiel) en mode ajout
  - Binary ⇨ Ouvre un fichier en accès binaire
  - Input ⇨ Ouvre un fichier texte (séquentiel) en mode lecture
  - Output ⇨ Ouvre un fichier texte (séquentiel) en mode écriture
  - Random ⇨ Ouvre un fichier en accès aléatoire
- *Access* désigne le type d'accès pour le mode binaire, tel que :
  - Read
  - Write
  - Read Write
- *Lock* désigne les restrictions d'accès pour les autres processus
  - Shared ⇨ tous les accès autorisés
  - Lock Read ⇨ Verrouillé en lecture
  - Lock Write ⇨ verrouillé en écriture
  - Lock Read Write ⇨ Verrouillé
- *Filenumber* est le numéro de fichier compris entre 1 et 255 pour les fichiers restreints et entre 256 à 511 pour les fichiers accessibles aux autres applications
- *Len* est la longueur des enregistrements en mode aléatoire ou éventuellement la taille du tampon en mode séquentiel

Présenté comme cela, ça a l'air un peu tordu, cependant vous allez voir que c'est assez simple.
Généralement on utilise des fichiers restreints à l'application ce qui fait qu'on ne précise pas la clause *Lock*.
Le paramètre *Len* n'est utilisé que pour les fichiers en mode aléatoire avec la taille de l'enregistrement.
Ce qui va nous ramener à quelques syntaxes, que nous pourrions lister comme suit :

1. Ouvrir un fichier texte en écriture

```
Open "d:\user\tutos\texte.txt" For Output As #1
```

2. Ouvrir un fichier texte en Ajout

```
Open "d:\user\tutos\texte.txt" For Append As #1
```

3. Ouvrir un fichier texte en lecture

```
Open "d:\user\tutos\texte.txt" For Input As #1
```

4. Ouvrir un fichier en mode aléatoire

```
Open "d:\user\tutos\random.dat" For Random As #1 Len=Len(TypeEnreg)
```

5. Ouvrir un fichier binaire en écriture

```
Open "d:\user\tutos\binaire.dat" For Binary Access Write As #1
```

6. Ouvrir un fichier binaire en lecture

```
Open "d:\user\tutos\binaire.dat For Binary Access Read As #1
```

Évidemment dans certains cas, il est intéressant de gérer le paramètre *Lock*, ce qui pourrait donner par exemple :

```
Open "d:\user\tutos\texte.txt" For Append Lock Write As #1
```

FreeFile

Renvoie le prochain numéro de fichier libre. L'argument passé éventuellement permet de forcer l'attribution d'un numéro de la zone restreinte ou de la zone ouverte.

FreeFile[(*rangenumber*)] **As Integer**

Si *rangenumber* est omis, le prochain numéro sera renvoyé sans tenir compte de la zone, s'il vaut 1 un numéro de la zone restreinte sera renvoyée (1 – 255), s'il vaut 2 ce sera un numéro de la zone ouverte (256 – 511).

Il est toujours plus prudent d'appeler la fonction FreeFile que d'utiliser un numéro en dur.

Close

Ferme le fichier désigné par le numéro passé comme argument.

EOF

Renvoie vrai quand la fin d'un fichier ouvert en mode séquentiel ou aléatoire est atteinte.

Attention, l'argument qui désigne le numéro de fichier n'attend pas le caractère #, on devra donc écrire EOF(1) et non EOF(#1).

LOF

Renvoie la taille en octets d'un fichier ouvert.

LOF(*filenumber As integer***) As Long**

Récupérer la taille du fichier permet de calculer le nombre d'enregistrements ou de dimensionner un tampon de réception.

Loc

Renvoie une valeur de type Long indiquant la position du dernier octet lu ou écrit en mode binaire, ou du dernier enregistrement lu ou écrit en mode aléatoire

Seek

Comme tout cela pouvait paraître trop simple, il existe une instruction **Seek** et une fonction **Seek**.

La fonction **Seek** fonctionne exactement de la même manière que la fonction **Loc** que nous venons de voir, l'instruction **Seek** permet de déplacer la position de lecture dans un fichier ouvert. De la forme :

Seek [**#**]*filenumber As integer*, *position As long*

Où *filenumber* est le numéro de fichier et *position* la position de lecture écriture du fichier ouvert.

Instructions d'écriture

Dans les fichiers texte, on utilise soit l'instruction **Print #**, soit l'instruction **Write #.**

Nous n'allons pas aborder ici l'utilisation de **Print #** car elle est plutôt utilisée pour les sorties formatées ce qui nous entraînerait un peu loin. La fonction **Write #** suit la forme :

Write #*filenumber*, [*outputlist*]

Où *filenumber* est le numéro de fichier et *outputlist* la liste des expressions à écrire dans le fichier séparées par une virgule.

Il y a des règles d'utilisation de la fonction **Write #** que vous pouvez obtenir en détail dans l'aide en ligne, mais ayez à l'esprit que :

Si *outputlist* est omis, l'instruction écrit une ligne vide dans le fichier, les valeurs numériques n'autorise que le point comme séparateur décimal, les chaînes sont entourées de guillemets et enfin que la fonction insère un saut de ligne à la fin du dernier élément de *outputlist*.

L'instruction Put est utilisée pour l'écriture des fichiers binaires et aléatoires. De la forme :

Put [**#**]*filenumber*, [*recnumber*], *varname*

Où *filenumber* est le numéro de fichier, *recnumber* un argument facultatif indiquant la position de l'opération d'écriture et *varname* la valeur à écrire.

Nous verrons des exemples d'utilisation de ces instructions dans les codes de l'étude de cas concluant ce chapitre.

Instructions de lecture

La fonction **Input #** est un peu la fonction inverse de **Write #**.

Input #*filenumber, varlist*

La fonction récupère dans le fichier des valeurs séparées par des virgules pour les affecter aux variables de l'argument *varlist*. Les variables doivent être correctement typées et les entrées du fichier correctement délimitées.

La fonction Line Input # lit le fichier par ligne pour affecter la ligne lue dans une variable de type String.

Line Input #*filenumber*, *varname*

La fonction Get lit la ou les données du fichier ouvert à partir de la position éventuellement précisée pour l'affecter à la variable passée en argument.

Get [**#**]*filenumber***,** [*recnumber*]**,** *varname*

En mode aléatoire, *recnumber* correspond au numéro d'enregistrement alors qu'en mode binaire il s'agit de l'octet.

Je vous invite vivement à lire l'aide si vous voulez manipuler des fichiers de type aléatoire.

Exemples

Nous allons ici nous livrer à quelques exemples de manipulations de fichiers dans divers mode afin de mieux comprendre le fonctionnement des accès aux fichiers. Notre fichier Excel est de la forme suivante :

	A	B	C	D	E	F	G	H	I	J
1	Cellule 21									
2	Date	Heure	T° 1	T° 2	T° 3	T° 4	Vitesse	Status	Polluants	RPAM
3	03/11/2002	14:40:16	40.6	21.5	37.4	40.7	2497	1	9	338400
4	03/11/2002	14:40:17	40.7	21.5	37.5	40.8	2500	0	9	338400
5	03/11/2002	14:40:18	41.2	21.5	37.6	41	2500	1	8	338400
6	03/11/2002	14:40:19	41.1	21.5	37.8	41.2	2500	1	8	338400
7	03/11/2002	14:40:20	41.5	21.5	37.9	41.3	2495	1	9	338400
8	03/11/2002	14:40:21	41.2	21.5	38.1	41.4	2500	0	8	338400
9	03/11/2002	14:40:22	41.8	21.5	38.2	41.6	2497	0	8	338400
10	03/11/2002	14:40:23	41.5	21.5	38.4	41.8	2492	1	8	338400
11	03/11/2002	14:40:24	42.3	21.5	38.4	41.9	2495	0	8	338400
12	03/11/2002	14:40:25	42.4	21.5	38.6	42.1	2495	0	8	338400
13	03/11/2002	14:40:26	42.5	21.5	38.8	42.2	2497	0	9	338400
14	03/11/2002	14:40:27	43	21.5	38.9	42.3	2500	0	8	338400
15	03/11/2002	14:40:28	43.2	21.5	39	42.5	2500	0	8	338400
16	03/11/2002	14:40:29	43.3	21.4	39.2	42.7	2500	0	8	338400
17	03/11/2002	14:40:30	43	21.5	39.3	42.8	2500	0	9	338400
18	03/11/2002	14:40:31	43.4	21.5	39.5	43	2497	1	8	338400
19	03/11/2002	14:40:32	43.7	21.5	39.6	43	2500	0	9	338400
20	03/11/2002	14:40:33	43.7	21.5	39.7	43.2	2497	0	9	338400
21	03/11/2002	14:40:34	43.9	21.5	39.9	43.3	2500	0	9	338400
22	03/11/2002	14:40:35	44	21.5	40	43.4	2497	0	9	338400
23	03/11/2002	14:40:36	44.3	21.5	40.1	43.6	2500	0	8	338400
24	03/11/2002	14:40:37	44.8	21.4	40.2	43.8	2497	1	9	338400
25	03/11/2002	14:40:38	44.7	21.5	40.3	43.9	2497	0	8	338400
26	03/11/2002	14:40:39	44.9	21.5	40.4	44.1	2497	0	9	338400
27	03/11/2002	14:40:40	44.9	21.4	40.6	44.2	2500	0	8	338400

Nous allons stocker et/ou relire ses données en utilisant ce que nous venons de voir.

Commençons par convertir ce fichier Excel en fichier texte.

Le code le plus simple serait :

```
Private Const Chemin As String = "D:\user\tutos\excel\"

Public Sub EcrireFichierTexte()

    Dim NumFichier As Integer, compteur As Long

    NumFichier = FreeFile
    Open Chemin & "texte.txt" For Output As #NumFichier
    For compteur = 3 To Range("data").Rows.Count + 2
        Write #NumFichier, Cells(compteur, 1).Value, Cells(compteur,
2).Value, Cells(compteur, 3).Value, Cells(compteur, 4).Value,
Cells(compteur, 5).Value, Cells(compteur, 6).Value, Cells(compteur,
7).Value, Cells(compteur, 8).Value, Cells(compteur, 9).Value,
Cells(compteur, 10).Value
    Next compteur
    Close #1

End Sub
```

Le code est simple à écrire, pas trop lisible et le fichier résultant encore moins, ce qui n'est pas bien grave si c'est pour l'utiliser avec Excel puisque comme nous le verrons dans l'étude du modèle objet Excel, il existe des méthodes intégrées pour lire directement ce type de fichier. Par contre l'écriture d'une fonction serait un peu plus complexe puisque s'il est possible d'affecter directement des valeurs de cellules dans l'instruction **Write #**, cela n'est pas autorisé avec **Input #**.

Il existe plusieurs façons de contourner ce problème. Globalement, il existe trois solutions de lecture du fichier, donnée par donnée, ligne par ligne ou lecture unique.

Regardons d'abord la première solution. Celle-ci repose sur le fait que nous connaissions globalement le format de fichier, notamment le fait qu'il y ait dix colonnes de valeurs. Nous pourrions alors écrire :

```
Public Sub LireFichierTexte()

    Dim NumFichier As Integer, cmptLigne As Long, cmptCol As Long, Recup As
Variant

    NumFichier = FreeFile
    Open Chemin & "texte.txt" For Input As #NumFichier
    With Worksheets("Feuil1").Range("A1")
        Do Until EOF(1)
            Input #NumFichier, Recup
            .Offset(cmptLigne, cmptCol).Value = Recup
            cmptCol = cmptCol + 1
            If cmptCol > 9 Then
                cmptCol = 0
                cmptLigne = cmptLigne + 1
            End If
        Loop
    End With
    Close #1

End Sub
```

Bien que fonctionnel, ce code présuppose un format ce qui est toujours un peu dangereux. Nous allons donc préférer une lecture globale, plus rapide qu'une approche par ligne et qui elle ne présuppose pas de format. Nous allons pour cela utiliser la fonction Split qui découpe des chaînes selon un séparateur. Cette fonction sera étudiée un peu plus loin lors de l'étude des fonctions de chaînes.

```
Public Sub LireFichierTexte()

    Dim NumFichier As Integer, TabLigne() As String, tabCol() As String,
Recup As String
    Dim cmpt1 As Long, cmpt2 As Long

    NumFichier = FreeFile
    Open Chemin & "texte.txt" For Binary Access Read As #NumFichier
    Recup = String(LOF(NumFichier), " ")
    Get #NumFichier, , Recup
    Close #NumFichier
    With Worksheets("Feuil1").Range("A1")
        TabLigne = Split(Recup, vbCrLf)
        For cmpt1 = 0 To UBound(TabLigne)
            tabCol = Split(TabLigne(cmpt1), ",")
            For cmpt2 = 0 To UBound(tabCol)
                .Offset(cmpt1, cmpt2).Value = tabCol(cmpt2)
            Next cmpt2
        Next cmpt1
    End With

End Sub
```

Si vous regardez les lignes de lecture du fichier, vous allez vous dire que je vous prends pour des jambons avec mon mode texte alors que j'effectue une lecture binaire du fichier. Pour pouvoir récupérer le fichier en une seule récupération, c'est pourtant la méthode la plus simple. On dimensionne un tampon de la taille du fichier et on charge tout le fichier dans ce tampon.

Il y a évidemment d'autres techniques d'écriture du fichier texte pour augmenter la lisibilité de celui-ci ou pour contourner les problèmes de séparateur mais ceci sort du cadre de notre cours.

Regardons maintenant le travail en mode aléatoire. Celui-ci n'a de sens que si vous pouvez obtenir une structure répétable dans les données à sauvegarder. Évidemment, c'est le cas dans notre exemple puisque chaque ligne peut être décrite dans un type utilisateur.

```
Public Type Enregistrement
    LaDate As Date
    Temp1 As Single
    Temp2 As Single
    Temp3 As Single
    Temp4 As Single
    Vitesse As Integer
    Status As Boolean
    Polluant As Byte
    RPAM As Long
End Type
```

Vous noterez que notre type n'a que neuf champs alors que le fichier à 10 colonnes. Pour des raisons de taille et de simplicité, nous avons tout intérêt à réunir les deux premiers champs en un seul puisque les deux informations peuvent être contenues dans le même champ.

Une fois le type créé, le code d'écriture est extrêmement simple.

```
Public Sub EcrireFichierAleatoire()

    Dim NumFichier As Integer, compteur As Long, MyEnr As Enregistrement

    NumFichier = FreeFile
    Open Chemin & "aleatoire.dat" For Random As #NumFichier Len =
Len(MyEnr)
    For compteur = 3 To Range("data").Rows.Count + 2
        With MyEnr
            .LaDate = Cells(compteur, 1).Value + Cells(compteur, 2).Value
            .Temp1 = Cells(compteur, 3).Value
            .Temp2 = Cells(compteur, 4).Value
            .Temp3 = Cells(compteur, 5).Value
            .Temp4 = Cells(compteur, 6).Value
            .Vitesse = Cells(compteur, 7).Value
            .Status = Cells(compteur, 8).Value
            .Polluant = Cells(compteur, 9).Value
            .RPAM = Cells(compteur, 10).Value
        End With
        Put #NumFichier, , MyEnr
    Next compteur
    Close #NumFichier

End Sub
```

Le code de relecture serait tout aussi simple.

```
Public Sub LireFichierAleatoire()

    Dim NumFichier As Integer, compteur As Long, MyEnr As Enregistrement
    Dim NbEnr As Integer

    NumFichier = FreeFile
    Open Chemin & "aleatoire.dat" For Random As #NumFichier Len =
Len(MyEnr)
    NbEnr = LOF(1) \ Len(MyEnr)
    For compteur = 1 To NbEnr
        Get #NumFichier, , MyEnr
        With Worksheets("Feuil1")
            .Cells(compteur, 1).Value = DateValue(MyEnr.LaDate)
            .Cells(compteur, 2).Value = TimeValue(MyEnr.LaDate)
            .Cells(compteur, 3).Value = MyEnr.Temp1
            .Cells(compteur, 4).Value = MyEnr.Temp2
            .Cells(compteur, 5).Value = MyEnr.Temp3
            .Cells(compteur, 6).Value = MyEnr.Temp4
            .Cells(compteur, 7).Value = MyEnr.Vitesse
            .Cells(compteur, 8).Value = MyEnr.Status
            .Cells(compteur, 9).Value = MyEnr.Polluant
            .Cells(compteur, 10).Value = MyEnr.RPAM
        End With
    Next compteur
    Close #NumFichier

End Sub
```

Notons toutefois que structurer les données pour le mode aléatoire n'a pas tellement de sens pour effectuer uniquement un travail de lecture / écriture dans Excel. Le mode aléatoire présente surtout un intérêt parce qu'il est possible de lire ou écrire facilement un enregistrement sans devoir agir sur tout le fichier.

L'exemple suivant va nous écarter du sujet de base, si vous débutez le VBA vous pouvez continuer à l'étude des fonctions d'information.

On utilise de moins en moins les fichiers sous forme binaire, pourtant cela peut présenter un intérêt certain. On l'utilise généralement pour diminuer la taille du fichier afin d'alléger la charge de sa communication, et parce qu'en l'absence du code de codage/décodage, les données qu'il contient sont inexploitables. Nous allons nous livrer ici à un petit exercice de style qui va nous permettre de manipuler un peu de code appliqué.

Le premier exercice comme souvent dans le développement va consister à observer et réfléchir. Pour pouvoir travailler efficacement dans le sens qui nous convient, en l'occurrence pour réduire la taille de stockage, il faut bien avoir à l'esprit la taille des types de données.

Il n'existe pas de spécifications pour l'écriture d'un fichier binaire. Vous pouvez utiliser n'importe quel principe de codage pour peu que vous soyez capable de décoder sans perte d'information. Les données n'ont pas de 'sens' dans un fichier binaire puisqu'on ne raisonne in fine que sur des octets.

Si nous observons le fichier exemple, la première constatation qui s'impose est que la date ne change pas et que chaque ligne est séparée d'une seconde de la précédente. Autrement dit, en connaissant la date et le temps de départ, l'écart de temps entre deux lignes et le nombre de ligne on peut déjà réécrire entièrement les deux premières colonnes.

Les quatre colonnes de températures suivantes présentent les mêmes caractéristiques. Il s'agit de chiffres décimaux à un chiffre après la virgule, relativement petits (<100). Si nous observons la colonne vitesse, nous constatons qu'il s'agit de nombres entiers variant peu ($\Delta < 15$). Enfin dans la dernière colonne, il s'agit d'entier relativement grand mais dont la précision n'est jamais inférieure à la centaine. Avec tout cela, nous allons pouvoir définir nos règles d'encodage.

Notre fichier va donc commencer par une ligne d'entête contenant :

- Le nombre de ligne (Integer)
- La date & l'heure du premier pas (Double)
- La fréquence des pas (Byte).
- Le diviseur des colonnes de température "T" (Byte)
- La valeur minimum du régime (Integer)
- Le multiplicateur RPAM (Byte).

L'écriture se fera ensuite par ligne, comme suit :

- Temp1, Temp2, Temp3, Temp4 corrigées (Integer)
- Δ vitesse (Byte)
- Polluants (Byte)
- RPAM corrigée (Integer)

Reste à traiter le cas de la voie Status. Celle-ci contient des valeurs booléennes, qui normalement se stockent sur deux octets. Cependant, comme nous l'avons vu avec les masques binaires, il est parfaitement possible de stocker huit valeurs booléennes dans un octet. La fin du fichier sera donc composée des valeurs de la colonne Status regroupées par 8.

Le code d'écriture du fichier va être :

```
Private Enum Masque
    Ligne1 = 1
    Ligne2 = 2
    Ligne3 = 4
    Ligne4 = 8
    Ligne5 = 16
    Ligne6 = 32
    Ligne7 = 64
    Ligne8 = 128
End Enum

Public Sub EcrireFichierBinaire()

    Dim NumFichier As Integer, compteur As Long
    Dim MinRegime As Integer, Status As Masque, NbLigne As Integer

    NumFichier = FreeFile
    Open Chemin & "binaire.dat" For Binary Access Write As #NumFichier
    '**************écriture de l'entête************
    'nombre de lignes
    NbLigne = CInt(Range("data").Rows.Count)
    Put #NumFichier, , CInt(Range("data").Rows.Count)
    'date de départ
    Put #NumFichier, , CDbl(Cells(3, 1).Value + Cells(3, 2).Value)
    'Fréquence
    Put #NumFichier, , CByte(1)
    'Diviseur Température
    Put #NumFichier, , CByte(10)
    'Regime minimum
    MinRegime = Application.WorksheetFunction.Min(Range("Data").Offset(,
6).Resize(, 1))
    Put #NumFichier, , MinRegime
    'Multiplicateur RPAM
    Put #NumFichier, , CByte(100)
    '**************fin écriture de l'entête************
    '**************écriture des données************
    For compteur = 3 To NbLigne + 2
        'températures *10
        Put #NumFichier, , CInt(Cells(compteur, 3).Value * 10)
        Put #NumFichier, , CInt(Cells(compteur, 4).Value * 10)
        Put #NumFichier, , CInt(Cells(compteur, 5).Value * 10)
        Put #NumFichier, , CInt(Cells(compteur, 6).Value * 10)
        'Delta regime
        Put #NumFichier, , CByte(Cells(compteur, 7).Value - MinRegime)
        'le status sera traité après
        'polluants
        Put #NumFichier, , CByte(Cells(compteur, 9).Value)
        'RPAM / 100
        Put #NumFichier, , CInt(Cells(compteur, 10).Value \ 100)
    Next compteur
    'début du traitement de la voie status, regroupement par 8 valeurs
```

```
For compteur = 3 To NbLigne + 2 Step 8
        With Cells(compteur, 8)
            If .Value Then Status = Status Or Ligne1
            If .Offset(1).Value Then Status = Status Or Ligne2
            If .Offset(2).Value Then Status = Status Or Ligne3
            If .Offset(3).Value Then Status = Status Or Ligne4
            If .Offset(4).Value Then Status = Status Or Ligne5
            If .Offset(5).Value Then Status = Status Or Ligne6
            If .Offset(6).Value Then Status = Status Or Ligne7
            If .Offset(7).Value Then Status = Status Or Ligne8
        End With
        Put #NumFichier, , CByte(Status)
        Status = 0
    Next compteur
    Close #NumFichier

End Sub
```

Et le code de lecture :

```
Private Enum Masque
    Ligne1 = 1
    Ligne2 = 2
    Ligne3 = 4
    Ligne4 = 8
    Ligne5 = 16
    Ligne6 = 32
    Ligne7 = 64
    Ligne8 = 128
End Enum

Public Sub LireFichierBinaire()

    Dim NumFichier As Integer, compteur As Long, RecupInt As Integer,
RecupByte As Byte
    Dim MinRegime As Integer, Status As Masque, NbLigne As Integer
    Dim Frequence As Byte, DivTemp As Byte, MultRPAM As Byte, DateDep As
Date

    NumFichier = FreeFile
    Open Chemin & "binaire.dat" For Binary Access Read As #NumFichier
    '**************lecture de l'entête************
    'nombre de lignes
    Get #NumFichier, , NbLigne
    'date de départ
    Get #NumFichier, , DateDep
    'Fréquence
    Get #NumFichier, , Frequence
    'Diviseur Température
    Get #NumFichier, , DivTemp
    'Regime minimum
    Get #NumFichier, , MinRegime
    'Multiplicateur RPAM
    Get #NumFichier, , MultRPAM
    '**************fin lecture de l'entête************
```

```
'**************lecture des données**************
    With Worksheets("Feuil1").Cells(1, 1)
        For compteur = 0 To NbLigne - 1
            'reconstruction du temps
            .Offset(compteur, 0).Value = DateValue(DateDep)
            .Offset(compteur, 1).Value = DateAdd("s", compteur * Frequence,
TimeValue(DateDep))
            'récupération des températures
            Get #NumFichier, , RecupInt
            .Offset(compteur, 2).Value = RecupInt / DivTemp
            Get #NumFichier, , RecupInt
            .Offset(compteur, 3).Value = RecupInt / DivTemp
            Get #NumFichier, , RecupInt
            .Offset(compteur, 4).Value = RecupInt / DivTemp
            Get #NumFichier, , RecupInt
            .Offset(compteur, 5).Value = RecupInt / DivTemp
            'récupération de la vitesse
            Get #NumFichier, , RecupByte
            .Offset(compteur, 6).Value = MinRegime + RecupByte
            'on ignore status pour l'instant
            'polluants
            Get #NumFichier, , RecupByte
            .Offset(compteur, 8).Value = RecupByte
            'RPAM
            Get #NumFichier, , RecupInt
            .Offset(compteur, 9).Value = CLng(RecupInt) * MultRPAM
        Next compteur
        For compteur = 0 To NbLigne - 1 Step 8
            Get #NumFichier, , RecupByte
            Status = RecupByte
            If Status And Ligne1 Then .Offset(compteur, 7).Value = 1 Else
.Offset(compteur, 7).Value = 0
            If Status And Ligne2 Then .Offset(compteur + 1, 7).Value = 1
Else .Offset(compteur + 1, 7).Value = 0
            If Status And Ligne3 Then .Offset(compteur + 2, 7).Value = 1
Else .Offset(compteur + 2, 7).Value = 0
            If Status And Ligne4 Then .Offset(compteur + 3, 7).Value = 1
Else .Offset(compteur + 3, 7).Value = 0
            If Status And Ligne5 Then .Offset(compteur + 4, 7).Value = 1
Else .Offset(compteur + 4, 7).Value = 0
            If Status And Ligne6 Then .Offset(compteur + 5, 7).Value = 1
Else .Offset(compteur + 5, 7).Value = 0
            If Status And Ligne7 Then .Offset(compteur + 6, 7).Value = 1
Else .Offset(compteur + 6, 7).Value = 0
            If Status And Ligne8 Then .Offset(compteur + 7, 7).Value = 1
Else .Offset(compteur + 7, 7).Value = 0
        Next compteur
    End With
    Close #NumFichier

End Sub
```

La taille du fichier binaire ainsi obtenu et cinq fois plus petite que celle du fichier texte précédemment vu, pour un traitement minimum.

Fonctions d'informations

Ces fonctions regroupent principalement les fonctions d'identification de sous type des variants ou de type d'une expression, décrites dans le tableau ci-dessous.

No fonction	*Argument*	Type renvoyé	Commentaire
IsArray	Variable	Booléen	Vrai si la variable est un tableau.
IsDate	Expression	Booléen	Vrai si l'expression peut être interprétée comme une date
IsEmpty	Expression	Booléen	Vrai si l'expression renvoie Empty ou si la variable n'a pas été initialisée
IsError	Expression	Booléen	Vrai si l'exression est une valeur d'erreur
IsMissing	Argument	Booléen	Vrai si l'argument fcultatif est manquant
IsNull	Expression	Booléen	Vrai si l'expression vaut Null
IsNumeric	Expression	Booléen	Vrai si l'expression peut être interprété comme un nombre
IsObject	Expression	Booléen	Vrai si l'expression est un objet
TypeName	Variable	String	Renvoie le nom du sous type de la variable
VarType	Variable	VbVarType(int)	Renvoie une constante entière selon le sous type de la variable

Fonctions de couleur

Les couleurs sont gérées selon le mode RGB (Red Green Blue), c'est-à-dire où chaque couleur est identifiée par une valeur entière définie par l'intensité relative de ses composantes rouges, vertes et bleues. Chaque composante peut prendre une valeur entre 0 et 255. On calcule à partir de cela une valeur entière (de type Long) telle que :
Valeur = Rouge + 256 * Vert + 65536 * Bleu
On représente souvent cette valeur en hexadécimal puisque du fait des décalages de 256, on peut voir les valeurs des composantes juste en lisant la valeur hexa.

QBColor

Renvoie une couleur prédéfinie en fonction de l'argument passé.
QBColor(*color As Integer*) **As Long**
L'argument *color* doit être compris entre 0 et 15.

QB	Long	Hex
0	0	0
1	8388608	800000
2	32768	8000
3	8421376	808000
4	128	80
5	8388736	800080
6	32896	8080
7	12632256	C0C0C0
8	8421504	808080
9	16711680	FF0000
10	65280	FF00
11	16776960	FFFF00
12	255	FF
13	16711935	FF00FF
14	65535	FFFF
15	16777215	FFFFFF

RGB

Renvoie une valeur de couleur en fonction des trois composantes passées en argument.
RGB(*red As Integer, green As Integer, blue As Integer*) **As Long**
Si un argument dépasse la valeur 255 il est ramené à cette valeur.

Fonctions d'interactions

Ces fonctions regroupent les fonctions d'interaction avec l'utilisateur, d'autres programmes ou le système. La plupart d'entre elles sortent du cadre de ce cours et ne seront pas étudiées ici. Éventuellement nous les détaillerons par la suite si nous en avons besoin.

Environ

Renvoie la valeur d'une des variables d'environnement du système.
Environ(***envstring***) **As Variant**
L'argument envstring est le nom de la variable désirée. Il peut être éventuellement un nombre qui représente la position de la variable dans la table d'environnement. Les variables peuvent différées d'un poste à l'autre, si une variable n'est pas trouvée, une chaîne de longueur nulle sera renvoyée.
Quelques exemples d'argument acceptés :

ALLUSERSPROFILE	APPDATA	CommonProgramFiles
COMPUTERNAME	ComSpec	FP_NO_HOST_CHECK
HOMEDRIVE	HOMEPATH	LOGONSERVER
NUMBER_OF_PROCESSORS	OS	Path
PATHEXT	PROCESSOR_ARCHITECTURE	PROCESSOR_IDENTIFIER
PROCESSOR_LEVEL	PROCESSOR_REVISION	ProgramFiles
SESSIONNAME	SystemDrive	SystemRoot
TEMP	TMP	USERDOMAIN
USERNAME	USERPROFILE	windir

InputBox

Affiche une boîte de saisie à l'utilisateur et renvoie la saisie éventuelle.
InputBox(***prompt***[, ***title***] [, ***default***] [, ***xpos***] [, ***ypos***]…) **As Variant**
Où *prompt* est le message affiché dans la boîte, *Title* son titre, *default* la valeur par défaut de la zone de saisie, *xpos* la position en abscisse d'affichage de la boîte et *ypos* la position en ordonnée.
Notez que si l'utilisateur clique sur le bouton annuler, la fonction renverra une chaîne vide. Nous n'irons pas plus loin de l'étude de cette fonction car nous verrons que le modèle Excel fournit une autre fonction InputBox plus intéressante.

MsgBox

Affiche une boîte de message à l'utilisateur.
MsgBox(***prompt***[, ***buttons***] [, ***title***] …) **As VbMsgBoxResult**
Où *prompt* est le message affiché dans la boîte, *Title* son titre. L'argument *buttons* est un masque binaire acceptant les éléments suivants :

Constante	Valeur	Description
vbOKOnly	0	Affiche le bouton **OK** uniquement.
vbOKCancel	1	Affiche les boutons **OK** et **Annuler**.
vbAbortRetryIgnore	2	Affiche le bouton **Abandonner**, **Réessayer** et **Ignorer**.
vbYesNoCancel	3	Affiche les boutons **Oui**, **Non** et **Annuler**.
vbYesNo	4	Affiche les boutons **Oui** et **Non**.
vbRetryCancel	5	Affiche les boutons **Réessayer** et **Annuler**.

Constante	Valeur	Description
vbCritical	16	Affiche l'icône **Message critique**.
vbQuestion	32	Affiche l'icône **Requête d'avertissement**.
vbExclamation	48	Affiche l'icône **Message d'avertissement**.
vbInformation	64	Affiche l'icône **Message d'information**.
vbDefaultButton1	0	Le premier bouton est le bouton par défaut.
vbDefaultButton2	256	Le deuxième bouton est le bouton par défaut.
vbDefaultButton3	512	Le troisième bouton est le bouton par défaut.
vbDefaultButton4	768	Le quatrième bouton est le bouton par défaut.

La réponse sera une constante entière de l'énumération *VbMsgBoxResult* telle que :

Constante	Valeur	
vbOK	**1**	**OK**
vbCancel	**2**	**Annuler**
vbAbort	**3**	**Abandonner**
vbRetry	**4**	**Réessayer**
vbIgnore	**5**	**Ignorer**
vbYes	**6**	**Oui**
vbNo	**7**	**Non**

Cette fonction permet donc d'afficher tous types de messages et d'orienter le code en fonction du bouton cliqué le cas échéant.

```
Public Sub TestMsg()

    Dim Reponse As VbMsgBoxResult

    MsgBox "Il fait beau", vbInformation + vbOKOnly, "Boîte 1"
    If MsgBox("voulez vous continuer ?", vbQuestion + vbYesNo, "Boîte 2") =
vbYes Then
        Reponse = MsgBox("Le lecteur n'est pas prêt", vbCritical +
vbAbortRetryIgnore, "Boîte 3")
        If Reponse = vbRetry Then
            MsgBox "c'est fini", vbOKCancel + vbDefaultButton2, "Boîte 4"
        End If
    End If

End Sub
```

Fonctions mathématiques

Fonctions standards

Toutes les fonctions que nous allons voir, sont des fonctions mathématiques standard qui attendent comme argument une expression renvoyant une valeur numérique et qui renvoient une valeur numérique. De la forme

NomFonction(*number*) **As TypeNumerique**

Nom de la fonction	*number*	Type renvoyé	Commentaire
Abs	tous type Num.	Id type *number*	Renvoie la valeur absolue de l'argument.
Atn	type Numérique	Double	Renvoie l'arctangente de l'argument (Radian)
Cos	Angle en radian	Double	Renvoie le cosinus de l'argument
Exp	>709,782712893	Double	Renvoie l'exponantielle de l'argument
Log	Double >0	Double	Renvoie le logarithme népérien de l'argument
Sgn	Double	Variant (Int)	Renvoie le signe de l'expression, -1 si négatif, 0 si nul et 1 si positif
Sin	Angle en radian	Double	Renvoie le Sinus de l'argument
Sqr	Double >=0	Double	Renvoie la racine carrée de l'argument
Tan	Angle en Radian	Double	Renvoie la tangente de l'argument

Fonctions spécifiques

Round

Round(*expression* [**,***numdecimalplaces*]) **As Variant**

Renvoie la valeur arrondie au nombre de décimales spécifié comme deuxième paramètre. Si celui-ci est omis ou vaut 0, le nombre sera arrondi à l'entier le plus proche. Par exemple :

```
Public Sub TestRound()

    Debug.Print Round(7.253, 2) '7.25
    Debug.Print Round(7.253, 1) '7.3
    Debug.Print Round(7.253, 0) '7
    Debug.Print Round(7.253) '7
    Debug.Print Round(-7.253, 2) '- 7.25
    Debug.Print Round(-7.253) '-7

End Sub
```

Tirage aléatoire, Randomize et Rnd

Rnd(*number*) **As Single**

La fonction renvoie un nombre aléatoire compris entre zéro et 1 basé sur la valeur de l'argument.

Si l'argument est positif ou omis, c'est le nombre aléatoire suivant dans la liste qui est renvoyée, si *number* est nul, c'est le dernier nombre aléatoire généré qui est envoyé sinon la fonction renvoie toujours le même nombre.

```
Public Sub TestRnd()

    Debug.Print Rnd(7.253) '0.7055475
    Debug.Print Rnd(7.253) '0.533424
    Debug.Print Rnd(0) '0.533424
    Debug.Print Rnd(-7.253) '0.7983214
    Debug.Print Rnd(-7.253) '0.7983214

End Sub
```

L'appel préalable de l'instruction Randomize sans argument invoque le générateur de nombre pseudo aléatoire.

Pour générer une valeur entière aléatoire dans une plage donnée, on utilise la formule :

```
Int((ValeurMax – ValeurMin + 1) * Rnd + ValeurMin)
```

Dans les deux cas les plus fréquent, vous trouverez parfois la notation :

```
Int((ValeurMax) * Rnd + 1)
```

Quand le limite basse vaut 1, et la notation

```
Int((ValeurMax + 1) * Rnd)
```

Quand la limite basse vaut 0.

```
Public Sub TestRnd()

Randomize
    'tirage de roulette (0 à 36)
    Debug.Print Int((36 + 1) * Rnd)
    'tirage de dé à 6 faces
    Debug.Print Int((6) * Rnd + 1)
    'tirage d'un jour entre Noël et le jour de l'an
    Debug.Print Int((31 – 25 + 1) * Rnd + 25)

End Sub
```

Fonctions de chaînes

Ces fonctions regroupent les fonctions de traitement des chaînes et les fonctions de conversion de caractère. Chaque fonction peut avoir entre une et six formes selon les cas qui suivent la nomenclature suivante :

Lorsque la fonction renvoie un variant, il existe généralement une seconde forme, notée NomFonction$ qui renvoie une chaîne. Ces formes sont généralement plus rapides à l'exécution que leurs homologues non typées. Lorsqu'il s'agit de fonction de caractères, il existe généralement une forme dite binaire sous la forme NomFonctionB et une forme Unicode notée NomFonctionW. Je noterai normalement toutes les formes dans la définition des fonctions.

Comparaison de chaînes

La comparaison de chaîne de caractères est normalement basée sur une comparaison dite 'binaire' dépendante de la représentation numérique des caractères. Ceci fait que la comparaison est sensible à la casse (Majuscule/Minuscule). Il existe cependant un autre type de comparaison, dite 'texte' insensibilisée à la casse. Toute les fonctions de chaîne nécessitant une opération de comparaison attendent donc un paramètre précisant le type de comparaison désirée. Ce paramètre n'est jamais obligatoire puisqu'il existe aussi une déclaration de niveau module, Option Compare, qui précise pour l'ensemble de ces fonctions le mode de comparaison lorsque l'argument est omis.

Cette instruction ne va pas affecter simplement les fonctions de chaînes mais aussi les opérateurs de comparaison. Le code ci-dessous illustre l'influence de l'instruction Option Compare.

```
Option Compare Binary

Public Sub InfluenceOption()

    If "test" <> "Test" Then
        MsgBox "Mode Option Compare Binary"
    Else
        MsgBox "Mode Option Compare Text"
    End If

End Sub
```

Si je change l'instruction pour

```
Option Compare Text
```

J'obtiendrais l'affichage du message correspondant.

De manière générale, on évite autant que faire se peut les comparaisons à l'aide d'opérateur pour privilégier l'utilisation de la fonction StrComp à des fins de lisibilité.

Traitement des caractères

Asc & Chr

Les caractères ont une représentation numérique, appelée code de caractère. Pendant assez longtemps, ce code a été codé sur un octet, donc compris entre 0 et 255, c'est le code ANSI. Comme il existe à travers le monde bien plus de 255 caractères, on a créé un nouveau code sur deux octets appelé Unicode.

La fonction Asc renvoie le code de caractère du caractère passé en argument, la fonction Chr renvoie le caractère correspondant au nombre passé en argument.

Asc(*string As String*) **As Integer**
AscB(*string As String*) **As Byte**
AscW(*string As String*) **As Integer**

L'argument string peut être une chaîne de plusieurs caractères, seul le code de la première lettre sera renvoyé.

Chr(*charcode As Long*) **As Variant**
Chr$(*charcode As Long*) **As String**
ChrB(*charcode As Byte*) **As Variant**
ChrB$(*charcode As Byte*) **As String**
ChrW(*charcode As Long*) **As Variant**
ChrW$(*charcode As Long*) **As String**

```
Private Function EnleveAccents(ByVal Chaîne As String) As String

    Dim compteur As Long, CarLu As String, ValRet As Integer
    For compteur = 1 To Len(Chaîne)
        CarLu = Mid(Chaîne, compteur, 1)
        ValRet = Asc(CarLu)
        Select Case ValRet
            Case 192 To 197: ValRet = 65
            Case 200 To 203: ValRet = 69
            Case 204 To 207: ValRet = 73
            Case 210 To 214: ValRet = 79
            Case 218 To 220: ValRet = 85
            Case 224, 226, 228: ValRet = 97
            Case 232 To 235: ValRet = 101
            Case 238, 239: ValRet = 105
            Case 244, 246: ValRet = 111
            Case 251, 252: ValRet = 117
        End Select
        EnleveAccents = EnleveAccents & Chr(ValRet)
    Next

End Function
```

Cette fonction enlève les caractères accentués de la chaîne passée en paramètre.

N.B : Normalement, on ne met qu'une instruction de code par ligne, le caractère de fin de ligne étant le retour chariot. Cependant, VBA interprète le caractère ":" (deux points) comme étant un séparateur d'instruction lorsqu'il est placé dans une ligne sans en être le dernier caractère. Cette notation quoique valide est généralement à éviter pour des raisons de lisibilité sauf quand le code est suffisamment trivial pour ne pas prêter à confusion.

Dans l'exemple ci-dessus, la notation :

```
Case 192 To 197: ValRet = 65
```

Équivaut à la notation

```
Case 192 To 197
    ValRet = 65
```

Recherche & Extraction

StrComp

La fonction StrComp compare les deux chaînes passées en argument selon le mode spécifié et renvoie un entier représentant le résultat de la comparaison.

StrComp(*string1* As String, *string2 As String*[, *compare As VbCompareMethod*]) **As Integer**

L'argument *compare* peut prendre les valeurs vbBinaryCompare (0) ou vbTextCompare (1). S'il est omis c'est l'instruction Option Compare qui fixe le mode, si celle-ci n'est pas déclarée, c'est une comparaison binaire qui sera faite. Le résultat renvoyé vaudra 0 si les deux chaînes sont identiques dans le mode de comparaison choisi, -1 si string1<string2, 1 dans le cas contraire.

```
Public Sub TriBulle(ByRef strArray() As String, Comparaison As
VbCompareMethod)

    Dim strTemp As String, cmpt As Long
    Dim Termine As Boolean, Encours As Boolean
    Termine = False

    Do While Termine = False
        For cmpt = LBound(strArray) To UBound(strArray) - 1
            If StrComp(strArray(cmpt), strArray(cmpt + 1), Comparaison) > 0
Then
                strTemp = strArray(cmpt)
                strArray(cmpt) = strArray(cmpt + 1)
                strArray(cmpt + 1) = strTemp
                Encours = True
            End If
        Next cmpt
        If Encours = False Then
            Exit Do
        End If
        Encours = False
    Loop


End Sub

Public Sub TestCompare()

Dim TabTest() As String
    TabTest = Split("Selon l'argument comparaison, le tri sera Sensible à
la casse", " ")
    Call TriBulle(TabTest, vbTextCompare)
    MsgBox Join(TabTest, " ")
    Call TriBulle(TabTest, vbBinaryCompare)
    MsgBox Join(TabTest, " ")

End Sub
```

Instr

La fonction **Instr** cherche la présence d'une chaîne dans une autre chaîne et renvoie sa position le cas échéant.

InStr([*start As Long*,]*string1 As String*, *string2 As String*[, *compare As VbCompareMethod*]) **As Long**

Où *start* est la position où la recherche commence, *string1* la chaîne dans laquelle on effectue la recherche, *string2* la chaîne recherchée et *compare* le mode de comparaison.

L'argument Compare définit le mode de comparaison et donc si la recherche est sensible ou non à la casse. Lorsque le paramètre *start* est omis, il est égal à 1 (début de la chaîne).

La position renvoyée est la position du premier caractère de la chaîne recherchée. Lorsque la recherche échoue la fonction renvoie 0 sauf dans les cas suivants :

Si *string1* ou *string2* est Null, la fonction renvoie Null.

Si *string2* est une chaîne vide, la fonction renvoie la valeur de *start*.

```
Public Function NbOccurence(ByVal ChaîneCible As String, Recherchee As
String) As Long

    Dim PosRech As Long

    PosRech = InStr(1, ChaîneCible, Recherchee, vbTextCompare)
    Do Until PosRech = 0
        NbOccurence = NbOccurence + 1
        PosRech = InStr(PosRech + Len(Recherchee) + 1, ChaîneCible,
Recherchee, vbTextCompare)
    Loop

End Fonction
```

Il existe une fonction InStrB qui renvoie la position de l'octet plutôt que la position du caractère.

Left, Mid & Right

Renvoie une partie de chaîne en fonction de critères de position. La fonction Left renvoie la partie gauche d'une chaîne, la fonction Right la partie droite et la fonction Mid une partie définie par les arguments qui lui sont passés. De la forme :

Left(*string As String, length As Long*) **As Variant**
LeftB(*string As String, length As Long*) **As Variant**
Left$(*string As String, length As Long*) **As String**
LeftB$(*string As String, length As Long*) **As String**

Où *string* est la chaîne dont les caractères de gauche seront renvoyés, *length* le nombre de caractères ou le nombre d'octets pour les variantes **LeftB**.

Right(*string As String, length As Long*) **As Variant**
RightB(*string As String, length As Long*) **As Variant**
Right$(*string As String, length As Long*) **As String**
RightB$(*string As String, length As Long*) **As String**

Où *string* est la chaîne dont les caractères de droite seront renvoyés, *length* le nombre de caractères ou le nombre d'octets pour les variantes **RightB.**

Mid(*string As String, start As Long*[, *length As Long*]) **As Variant**
MidB(*string As String, start As Long*[, *length As Long*]) **As Variant**
Mid$(*string As String, start As Long*[, *length As Long*]) **As String**
MidB$(*string As String, start As Long[, length As Long]*) **As String**

Où *string* est la chaîne dont les caractères seront renvoyés, *start* la position du premier caractère (octet pour **MidB**) à renvoyer, *length* le nombre facultatif de caractères ou d'octets (pour **MidB**) devant être renvoyés.

Nous avons vu un exemple classique d'utilisation de la fonction **Mid** lors de l'étude de **Asc** pour le parcours d'une chaîne caractère par caractère.

```
Public Function MyReplace(ByVal Chaîne As String, ByVal Cherche As String,
ByVal Remplace As String) As String

    Dim PosRech As Long

    PosRech = InStr(1, Chaîne, Cherche, vbTextCompare)
    Do Until PosRech = 0
        Chaîne = Left(Chaîne, PosRech - 1) & Remplace & Right(Chaîne,
Len(Chaîne) - PosRech - Len(Cherche) + 1)
        PosRech = InStr(1, Chaîne, Cherche, vbTextCompare)
    Loop
    MyReplace = Chaîne

End Function
```

Len

Renvoie la longueur de la chaîne de caractères passée en argument, ou la taille de la variable si ce n'est pas une chaîne. Existe sous forme LenB pour une réponse en octets.

Len(*expression*) **As Long**

LenB(*expression*) **As Long**

InStrRev

Fonctionne comme la fonction InStr mais en partant de la fin de la chaîne.

InstrRev(*stringcheck As String*, *stringmatch As String*[, *start As Long*[, *compare As VbCompareMethod*]]) **As Long**

Où *stringcheck* est la chaîne dans laquelle est effectuée la recherche et *stringmatch* la chaîne cherchée.

Si *position* est omis ou égal à -1, la recherche commence à la fin de la chaîne.

```
Private Function ExtractParenthese(ByVal Formule As String) As Variant

    Dim PosFerme As Long, PosOuvert As Long
    Dim cmpt As Long, TabReponse() As String
    PosFerme = InStr(1, Formule, ")", vbTextCompare)
    Do Until PosFerme = 0
        PosOuvert = InStrRev(Formule, "(", PosFerme, vbTextCompare)
        ReDim Preserve TabReponse(0 To cmpt)
        TabReponse(cmpt) = Mid(Formule, PosOuvert + 1, PosFerme - PosOuvert
- 1)
        cmpt = cmpt + 1
        Formule = Left(Formule, PosOuvert - 1) & IIf(PosFerme <
Len(Formule), Mid(Formule, PosFerme + 1), "")
        PosFerme = InStr(1, Formule, ")", vbTextCompare)
    Loop
    If Len(Formule) > 0 Then
         ReDim Preserve TabReponse(0 To cmpt)
         TabReponse(cmpt) = Formule
    End If
    ExtractParenthese = TabReponse

End Function
```

Split

La fonction Split sépare une chaîne selon un séparateur et renvoie un tableau (de base 0) des sous chaînes correspondantes.

Split(*expression* As String[**,** *delimiter* As String[**,** *limit As Long*[**,***compare As VbCompareMethod*]]]) **As String()**

Où *expression* est la chaîne à séparer, *delimiter* le séparateur et *limit* le nombre de sous chaînes à renvoyer.

Si le séparateur est de longueur nulle ou s'il n'est pas présent dans la chaîne, il sera renvoyé un tableau d'un élément contenant toute la chaîne. S'il est omis, c'est le caractère espace qui sera le séparateur.

Si *limit* est omis, toutes les sous chaînes seront renvoyées.

```
Public Sub TestSplit()
    Dim strTest As String, msg As String, cmpt As Long, TabSplit() As
String
    strTest = "un, deux, trois"
    TabSplit = Split(strTest, ",")
    For cmpt = LBound(TabSplit) To UBound(TabSplit)
        msg = msg & "[élément " & cmpt & " = " & TabSplit(cmpt) & "] "
    Next cmpt
    Debug.Print msg
    '[élément 0 = un] [élément 1 =  deux] [élément 2 =  trois]
    msg = ""
    TabSplit = Split(strTest, ",", 2)
    For cmpt = LBound(TabSplit) To UBound(TabSplit)
        msg = msg & "[élément " & cmpt & " = " & TabSplit(cmpt) & "] "
    Next cmpt
    Debug.Print msg
    '[élément 0 = un] [élément 1 =  deux, trois]
    msg = ""
    TabSplit = Split(strTest, ";")
    For cmpt = LBound(TabSplit) To UBound(TabSplit)
        msg = msg & "[élément " & cmpt & " = " & TabSplit(cmpt) & "] "
    Next cmpt
    Debug.Print msg
    '[élément 0 = un, deux, trois]
    msg = ""
End Sub
```

Filter

La fonction Filter compare une chaîne à un tableau de chaînes et renvoie un tableau contenant tout les éléments comprenant ou excluant la chaîne spécifiée.

Filter(*sourcearray As string()***,** *match As String*[**,** *include As Boolean*[**,***compare As VbCompareMethod*]]) **As String()**

Sourcearray doit être un tableau à une dimension.

Si *include* est vrai, tous les éléments contenant la chaîne *match* seront renvoyés sinon ce sera tous les éléments ne la contenant pas. S'il n'y a pas de correspondance, un tableau vide sera renvoyé.

```
Public Sub TestFilter()
    Dim TabString() As String, TabReponse As Variant
    Dim Chaîne As String
    Chaîne = "C'est un test et comme tous les tests il peut échouer"
    TabString = Split(Chaîne, " ")
    TabReponse = Filter(TabString, "test", True, vbTextCompare)
    If UBound(TabReponse) > -1 Then
        MsgBox UBound(TabReponse) + 1 '2
    End If
End Sub
```

Modification

LTrim, RTrim & Trim

Les fonctions Trim renvoient une chaîne sans les espaces situés à une ou aux deux extrémités de celle-ci.

La fonction LTrim enlève les espaces de gauches, RTrim ceux de droites et Trim des deux cotés.

Function LTrim(*String As String)* **As Variant**
Function LTrim$(*String As String)* **As String**
Function RTrim(*String As String)* **As Variant**
Function RTrim$(*String As String)* **As String**
Function Trim(*String As String)* **As Variant**
Function Trim$(*String As String)* **As String**

Replace

Remplace une ou plusieurs occurrences d'une chaîne contenue dans une chaîne par une autre chaîne.

Replace(*expression As String*, *find As String*, *replace As String*[, *start As Long*[, *count As Long*[,*compare As VbCompareMethod*]]]) **As String**

Où *expression* est la chaîne dans laquelle les remplacements sont effectués, *find* la chaîne à rechercher, *replace* la chaîne de remplacement, *start* le caractère de départ et *count* le nombre maximum de remplacement.

Si *start* est omis, il vaut 1, s'il est supérieur à 1, tous les caractères dont la position est inférieure à start seront enlevés de la chaîne renvoyée.

Si *count* est omis, il vaut -1 c'est-à-dire que tous les remplacements possibles seront effectués.

```
Public Sub TestReplace()

    Dim TabString() As String, TabReponse As Variant
    Dim Chaîne As String

    Chaîne = "C'est un test et comme tous les tests il peut échouer"
    Debug.Print Replace(Chaîne, "test", "essai", 1, -1, vbTextCompare)
    'C'est un essai et comme tous les essais il peut échouer
    Debug.Print Replace(Chaîne, "test", "essai", 1, 1, vbTextCompare)
    'C'est un essai et comme tous les tests il peut échouer
    Debug.Print Replace(Chaîne, "test", "essai", 15, -1, vbTextCompare)
    'et comme tous les essais il peut échouer

End Sub
```

LCase & Ucase

Convertit la chaîne dans la casse spécifiée. LCase convertira la chaîne en minuscule, UCase en majuscule.

LCase(*string As String*) **As Variant**
LCase$(*string As String*) **As String**
UCase(*string As String*) **As Variant**
UCase$(*string As String*) **As String**

StrConv

Convertit la chaîne passée en argument selon le mode spécifié.

StrConv(*string As String, conversion As vbStrConv,…*) **As String**

Où *string* est la chaîne à convertir et *conversion* le masque binaire de conversion. Celui-ci accepte la composition des valeurs suivantes :

Constante	Valeur	Description
vbUpperCase	**1**	Convertit la chaîne en majuscules (Eqv UCase)
vbLowerCase	**2**	Convertit la chaîne en minuscules. (Eqv LCase)
vbProperCase	**3**	Convertit la première lettre de chaque mot de la chaîne en majuscule.
vbUnicode	**64**	Convertit la chaîne en Unicode à l'aide de la page de code par défaut du système.
vbFromUnicode	**128**	Convertit la chaîne Unicode dans la page de code par défaut du système.

StrReverse

Renvoie une chaîne dont les caractères sont en ordre inverse de la chaîne passée en argument.

Construction

Join

C'est la fonction inverse de Split, elle transforme un tableau de chaînes en une chaîne, éventuellement avec un séparateur.

Join(*sourcearray As String()*[, *delimiter As String*]) **As String**

Sourcearray doit être un tableau à une dimension. Si *delimiter* est omis, le caractère espace sera utilisé, si c'est une chaîne vide, les éléments du tableau seront concaténés.

Le code suivant remplace les virgules par des tabulations dans un fichier texte.

```
Public Sub ConvertFichierTexte()

    Dim NumFichier As Integer, TabLigne() As String, Recup As String
    Dim cmpt As Long

    NumFichier = FreeFile
    Open Chemin & "texte.txt" For Binary Access Read As #NumFichier
    Recup = String(LOF(NumFichier), " ")
    Get #NumFichier, , Recup
    Close #NumFichier
    TabLigne = Split(Recup, vbCrLf)
    For cmpt = 0 To UBound(TabLigne)
        TabLigne(cmpt) = Join(Split(TabLigne(cmpt), ","), vbTab)
    Next cmpt
    Recup = Join(TabLigne, vbCrLf)
    NumFichier = FreeFile
    Open Chemin & "txtConv.txt" For Binary Access Write As #NumFichier
    Put #NumFichier, , Recup
    Close #NumFichier

End Sub
```

Space

La fonction Space renvoie une chaîne composée d'espaces. De la forme :

Function Space(*Number As Long*) **As Variant**
Function Space$(*Number As Long*) **As String**
Où *number* représente le nombre d'espaces composant la chaîne.
On utilise généralement cette fonction lorsqu'on procède à des appels système qui nécessite une chaîne dite 'tampon' pour la réponse.
L'exemple suivant récupère le login de l'utilisateur en cours.

```
Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (ByVal
lpBuffer As String, nSize As Long) As Long

Public Function UtilConnect() As String

    Dim Tampon As String, retour As Long, Taille As Long

        Taille = 255
        Tampon = Space(Taille)
        retour = GetUserName(Tampon, Taille)
        UtilConnect = Left$(Tampon, Taille - 1)

End Function
```

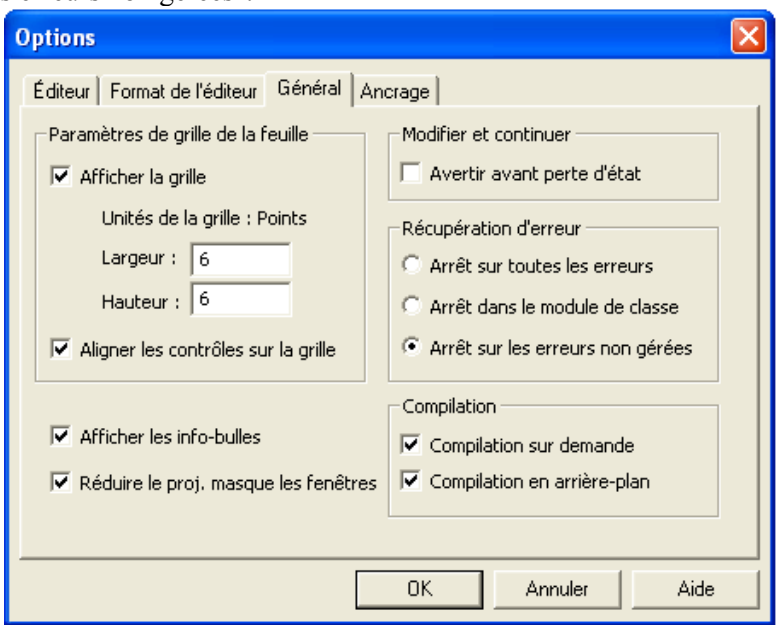
String

Sensiblement identique à la fonction précédente, renvoie une chaîne composée d'un caractère passé en paramètre répété le nombre de fois défini par l'argument correspondant.
String(*number As Long*, *character*) **As Variant**
String$(*number As Long*, *character*) **As String**
Où *number* représente le nombre de fois où *character* sera répété. *character* peut être un code de caractère compris entre 0 et 255, si il est supérieur, il sera utilisé sous la forme *character mod 256*. Il peut être aussi une chaîne dont seul le premier caractère sera utilisé.

Gestion des erreurs

Bien que les concepts soient relativement simples, la gestion des erreurs va s'avérer comme la partie la plus complexe que nous verrons dans ce cours. L'écriture d'un code de gestion d'erreurs de bonne qualité n'est pas une tâche aisée contrairement aux apparences.

En VBA, la gestion des erreurs récupérables se fait par le biais de l'instruction **On Error**. Le fait d'utiliser cette instruction n'est pas nécessairement suffisant pour que la récupération se fasse correctement. Vous devrez parfois configurer votre environnement de développement. Pour cela, allez dans le menu "Outils – Options" de l'éditeur de code, sélectionnez l'onglet "Général" puis sélectionnez l'option "Arrêt sur les erreurs non gérées".



Il existe grosso modo deux façons de gérer les erreurs même si on peut envisager dans certains scénarii plus de variantes, le traitement en place et le traitement centralisé.

L'erreur récupérable est représentée par l'objet VBA ErrObject, qu'on manipule au travers de la fonction Err. Cet objet expose plusieurs propriétés, dont :

- Description → Chaîne décrivant l'erreur
- Number → Numéro de l'erreur
- Source → Chaîne identifiant l'émetteur de l'erreur

Et deux méthodes

- Clear → Réinitialise l'objet ErrObject
- Raise → Lève une erreur récupérable

Les propriétés sont généralement valorisées par Visual Basic, cependant vous pouvez être vous-même générateur d'une erreur grâce à la méthode Raise. Prenons l'exemple suivant :

```
Private Const PI = 3.14159265

Private Function CalcCylindree(ByVal Alesage As Single, ByVal Course As
Single, ByVal NbCylindre As Byte) As Single

    If Sgn(Alesage) = -1 Or Sgn(Course) = -1 Then
        Err.Raise 5, "CalcCylindree", "Les valeurs d'alésage et de course
doivent être strictement positives"
    ElseIf NbCylindre = 0 Then
        Err.Raise 5, "CalcCylindree", "Le Nombre de cylindre ne peut pas
être nul"
    End If
    CalcCylindree = (PI * Course * Alesage / 2) / 4 / 10 * NbCylindre

End Function
```

Dans cette fonction, je vais générer une erreur si un des arguments n'est pas strictement positif. On peut évidemment se demander quel est l'intérêt d'un tel code, puisqu'en l'occurrence je pourrais traiter le problème dans le corps de la fonction. Cependant en procédant ainsi, je laisse la possibilité à celui qui programme la procédure appelante de gérer l'erreur comme il le souhaite.

Dans ce cas, je génère l'erreur 5 qui par convention est "Appel de procédure ou argument incorrect"[4]. J'utilise les arguments de la méthode Raise pour personnaliser la description et la source, mais je pourrais écrire aussi :

```
Err.Raise 5
```

Comme 5 est un numéro d'erreur prédéfinie, la valeur de la description serait le message par défaut de l'erreur, et la source serait "VBAProject" qui est la valeur par défaut de la source quand une erreur se produit dans un module standard. Il n'y a pas d'obligation à utiliser des erreurs prédéfinies. Vous pouvez très bien utiliser des numéros d'erreur personnalisé (par convention >1000) ce qui peut être utile pour le code de gestion d'erreurs.

Traitement centralisé

Soyons clair, bien que centralisé, il n'en est pas moins immédiat puisque la levée d'une erreur déclenche automatiquement l'exécution du code de traitement d'erreurs. Ce traitement utilise les étiquettes de lignes, c'est-à-dire la localisation d'un point spécifique du code grâce à une étiquette.

Celles-ci suivent les règles suivantes :

- Elles sont composées de n'importe quelle combinaison de caractères commençant par une lettre et se terminant par le signe deux-points (**:**)
- Elles ne distinguent pas les majuscules des minuscules
- Elles doivent commencer au niveau de la première colonne (le plus à gauche dans l'éditeur)

L'instruction de gestion d'erreurs sera alors de la forme **On Error Goto** *Etiquette*. Généralement, on place le code de gestion d'erreurs en fin de procédure. On fait donc précéder la ligne de l'étiquette d'une instruction de sortie de procédure (Exit Sub ou Exit Function selon le cas) pour que le code d'erreur ne s'exécute pas en cas de fonctionnement normal.

Pour que les explications soient plus claires, partons de l'exemple suivant :

```
Private Const PI = 3.14159265
```

```
Private Function CalcCylindree(ByVal Alesage As Single, ByVal Course As
Single, ByVal NbCylindre As Byte) As Single

    If Sgn(Alesage) < 1 Or Sgn(Course) < 1 Then
        Err.Raise 5 ', "Cylindree", "Les valeurs d'alésage et de course
doivent être strictement positives"
    ElseIf NbCylindre = 0 Then
        Err.Raise 5 ', "Cylindree", "Le Nombre de cylindre ne peut pas être
nul"
    End If
    CalcCylindree = (PI * Course * Alesage ^ 2) / 4 / 1000 * NbCylindre

End Function

Private Function CalcRapVol(ByVal Cylindree As Single, ByVal VolChambre As
Single) As Single

    CalcRapVol = (Cylindree + VolChambre) / VolChambre

End Function
```

[4] Vous trouverez la liste des erreurs récupérables dans l'aide en ligne

```
Public Sub CalculMoteur()

Dim Ales As Single, Course As Single, NCyl As Integer, VolumeC As Single
Dim Cylindree As Single, RapportVolu As Single

    On Error GoTo TraiteErreur
    Ales = Application.InputBox("Entrer la valeur d'alésage en mm",
"Saisie", Type:=1)
    Course = Application.InputBox("Entrer la valeur de la course en mm",
"Saisie", Type:=1)
    NCyl = Application.InputBox("Entrer le nombre de cylindre", "Saisie",
Type:=1)
    VolumeC = Application.InputBox("Entrer le volume de la chambre de
combustion en cm^3", "Saisie", Type:=1)
    Cylindree = CalcCylindree(Ales, Course, NCyl)
    RapportVolu = CalcRapVol(Cylindree, VolumeC)
    MsgBox Cylindree & vbNewLine & RapportVolu
    Exit Sub
TraiteErreur:
    If Err.Number = 5 Then
        If NCyl = 0 Then
            NCyl = Application.InputBox("Le nombre de cylindre doit être
positif", "Erreur", 1, Type:=1)
        ElseIf Sgn(Ales) < 1 Then
            If Ales = 0 Then
                Ales = Application.InputBox("L'alésage doit être
strictement positif", "Erreur", 1, Type:=1)
            Else
                Ales = -1 * Ales
            End If
        Else
            If Course = 0 Then
                Course = Application.InputBox("La course doit être
strictement positive", "Erreur", 1, Type:=1)
            Else
                Course = -1 * Course
            End If
        End If
        Resume
    End If

End Sub
```

Observons la procédure 'CalculMoteur'. Nous activons la récupération d'erreur avec la première instruction. Celle-ci signifie donc que si une erreur se produit dans le code, le curseur d'exécution atteindra la ligne 'TraiteErreur' et le code de traitement sera exécuté. Si le code se déroule sans encombre, la procédure se terminera sur l'instruction Exit Sub (et non sur End Sub) et le code ne sera pas traité.

Regardons maintenant le code de traitement des erreurs. Si une erreur se produit, la propriété Number de ErrObject va être différente de 0.

Nous allons alors tester le numéro de l'erreur. Si celui-ci est égal à 5, c'est que nous avons levé l'exception de la fonction 'CalcCylindree', c'est-à-dire qu'un des trois arguments n'est pas strictement positif. Nous allons donc appliquer la stratégie suivante, si l'argument faux est négatif, je vais le multiplier par -1, s'il est nul, nous allons redemander la saisie d'une valeur pour cet argument. Puis nous allons tenter de ré exécuter la ligne qui a levé l'erreur.

Pour atteindre cette ligne, nous utiliserons l'instruction Resume.

L'instruction **Resume** sert à retourner dans le code qui a déclenché l'erreur. On l'utilise sous la forme :

Resume → Retourne à la ligne qui a déclenchée l'erreur

Resume Next → Retourne à la ligne suivant celle qui a levée l'erreur

Resume ***Etiquette*** → Va à la ligne de l'étiquette.

Évitez d'utilisez Resume Etiquette qui peut rapidement compliquer la lecture de votre code.

N.B : L'instruction Resume réinitialise l'objet ErrObject, comme si on faisait Err.Clear.

Attention, Resume atteint la ligne qui a déclenchée l'erreur dans la procédure qui contient le code de gestion d'erreur. Dans notre cas,

```
Cylindree = CalcCylindree(Ales, Course, NCyl)
```

Supposons maintenant qu'au lieu d'un argument faux, il y en ait deux, que va-t-il se passer?

Le code va entrer dans le code de gestion d'erreur une première fois et traiter la première erreur dans l'ordre des conditions, puis l'instruction Resume va remettre le curseur d'exécution sur la ligne d'appel de la fonction 'CalcCylindree'. Une erreur va de nouveau se produire puisque notre code de traitement d'erreur ne corrige qu'un argument à la fois. Cette nouvelle erreur va renvoyer le code dans le traitement d'erreur pour traiter le second argument. Notre gestion d'erreur est donc bien écrite.

Où du moins l'est-il pour le traitement de l'erreur explicitement levée par la fonction 'CalcCylindree'. Imaginons que notre opérateur rêveur entre 300 comme nombre de piston. Il ne se passera rien à l'écran et rien ne sera calculé.

En fait, cette erreur de saisie va déclencher une erreur de dépassement de capacité puisque l'argument 'NbCylindre' est de type Byte qui ne peut pas excéder 255. Cette erreur est l'erreur 6. Le code d'erreur va alors être exécuté, mais tel qu'il est écrit, il ne traite que l'erreur 5 et c'est pour cela qu'en apparence il ne se passe rien. Pour parer à cette éventualité nous allons donc modifier notre code d'erreur tel que :

```
TraiteErreur:
    Select Case Err.Number
        Case 5
            If NCyl = 0 Then
                NCyl = Application.InputBox("Le nombre de cylindre doit
être positif", "Erreur", 1, Type:=1)
            ElseIf Sgn(Ales) < 1 Then
                If Ales = 0 Then
                    Ales = Application.InputBox("L'alésage doit être
strictement positif", "Erreur", 1, Type:=1)
                Else
                    Ales = -1 * Ales
                End If
            Else
                If Course = 0 Then
                    Course = Application.InputBox("La course doit être
strictement positive", "Erreur", 1, Type:=1)
                Else
                    Course = -1 * Course
                End If
            End If

        Case 6
            NCyl = Application.InputBox("Le nombre de cylindre doit être
compris entre 1 et 255", "Erreur", 1, Type:=1)

    End Select
    Resume
```

Nous voilà donc avec un code de gestion d'erreur prêt à tout. Et bien non, pas encore. Imaginons maintenant que ce diable d'utilisateur entre 0 comme valeur de volume. La fonction de calcul de la cylindrée va bien se passer, puis le curseur d'exécution entre dans la fonction 'CalcRapVol'. Le calcul de cette fonction va lever une erreur puisqu'il y aura une division par zéro. Cette fonction ne possède pas de gestionnaire d'erreur et pourtant l'exécution ne va pas s'arrêter.

C'est la le deuxième concept à appréhender correctement. Dés lors qu'il y a un gestionnaire d'erreur actif, les erreurs vont remonter le courant, comme les saumons, sauf que dans le cas des erreurs c'est la pile des appels.

Le mécanisme est le suivant. L'erreur va d'abord être traitée au niveau de la procédure où elle se produit. S'il n'y a pas de gestionnaire d'erreur dans celle-ci, il va remonter l'erreur au gestionnaire de la procédure appelante et ainsi de suite jusqu'à trouver un gestionnaire actif. Ce mécanisme n'est pas inintéressant mais il est extrêmement périlleux.

En effet, si vous avez prévu cette remontée, pas de problème. Mais si il s'est produit une erreur que vous n'aviez pas prévue, vous allez déclencher un traitement qui n'est probablement pas adapté en plus de rendre plus difficile la détection de l'erreur.

Nous aurions trois possibilités dans notre exemple, désactiver le gestionnaire après le calcul de la cylindrée, traiter la division par zéro au niveau de la fonction de calcul du rapport volumique ou la traiter au niveau du gestionnaire de la procédure.

Pour désactiver le gestionnaire d'erreur, on utilise l'instruction On Error Goto 0.

Ce qui donnerais un code tel que

```
Public Sub CalculMoteur()

Dim Ales As Single, Course As Single, NCyl As Integer, VolumeC As Single
Dim Cylindree As Single, RapportVolu As Single

    On Error GoTo TraiteErreur
    Ales = Application.InputBox("Entrer la valeur d'alésage en mm",
"Saisie", Type:=1)
    Course = Application.InputBox("Entrer la valeur de la course en mm",
"Saisie", Type:=1)
    NCyl = Application.InputBox("Entrer le nombre de cylindre", "Saisie",
Type:=1)
    VolumeC = Application.InputBox("Entrer le volume de la chambre de
combustion en cm^3", "Saisie", Type:=1)
    Cylindree = CalcCylindree(Ales, Course, NCyl)
    On Error GoTo 0
    RapportVolu = CalcRapVol(Cylindree, VolumeC)
    MsgBox Cylindree & vbNewLine & RapportVolu
    Exit Sub
TraiteErreur:
    . . . . .
```

Sinon la modification du gestionnaire de la procédure serait :

```
TraiteErreur:
    Select Case Err.Number
        Case 5
            If NCyl = 0 Then
                NCyl = Application.InputBox("Le nombre de cylindre doit
être positif", "Erreur", 1, Type:=1)
            ElseIf Sgn(Ales) < 1 Then
                If Ales = 0 Then
                    Ales = Application.InputBox("L'alésage doit être
strictement positif", "Erreur", 1, Type:=1)
                Else
                    Ales = -1 * Ales
                End If
            Else
                If Course = 0 Then
                    Course = Application.InputBox("La course doit être
strictement positive", "Erreur", 1, Type:=1)
                Else
                    Course = -1 * Course
                End If
            End If

        Case 6
            NCyl = Application.InputBox("Le nombre de cylindre doit être
compris entre 1 et 255", "Erreur", 1, Type:=1)
```

```
        Case 11
            VolumeC = Application.InputBox("Entrer un volume supérieur à 0
pour la chambre de combustion en cm^3", "Saisie", Type:=1)

    End Select
    Resume
```

Il n'est pas toujours facile de savoir quel choix prendre. En théorie, lorsqu'on commence à écrire du code de gestion d'erreur, on cherche à traiter toutes les erreurs sauf lorsqu'on fait du traitement immédiat. Cependant la remontée des erreurs dans un gestionnaire situé assez loin dans la pile des appels peut avoir des effets de bord non négligeable. L'instruction Resume ne renverra pas le curseur d'exécution au niveau de la ligne en erreur mais au niveau de l'appel ayant déclenché celle-ci. Si lors des appels de fonction vous avez utilisé des variables globales ou des variables statiques, leurs valeurs ont pu être changées avant le déclenchement de l'erreur, et un deuxième parcours de la pile des appels aura un contexte différent et probablement un résultat différent. C'est généralement pour cela que lorsqu'on écrit du code de gestion d'erreur, on essaye de l'écrire assez près de la source de l'erreur et qu'on écrit des petits gestionnaires dans les procédures appelées plutôt qu'un grand dans la procédure principale.

Dans notre exemple cependant, le traitement de la division par 0 dans le code de la procédure appelante ne pose pas de problème, et nous avons enfin écrit un code robuste où les erreurs sont gérées correctement.

Et pourtant ce n'est pas vraiment le cas, car si nous avons bien traités les erreurs, notre procédure est boguée. Imaginons que notre utilisateur distrait commence à exécuter son programme. Tout à coup, il se rend compte qu'il a oublié la valeur du volume de la chambre de combustion sur son banc. Comme c'est un utilisateur expérimenté, il sait que tout trajet qui croise la machine à café peut s'avérer nettement plus long que le temps normalement prévu pour le parcourir. Il décide donc de cliquer sur le bouton 'Annuler' quand on lui demande le diamètre de l'alésage. Et la quelle ne sera pas sa surprise de voir que le programme va lui demander le nombre de cylindres. Que se passe-t-il donc ?

La méthode InputBox de l'objet Application renvoie la valeur saisie si vous cliquez sur 'OK' et False si vous cliquez sur 'Annuler'. En cliquant sur 'Annuler', le programme va tenter d'affecter la valeur False à la variable 'Ales'. Il y a là une incompatibilité de type, ce qui va lever une erreur de type 6, celle-ci va déclencher le code de gestion d'erreur, et le traitement de l'erreur numéro 6 provoque l'affichage d'une boîte de saisie pour le nombre de cylindre.

Dans cet exemple, le bug est relativement évident, mais ce n'est pas toujours le cas et parfois il est assez ardu de retrouver le bug lorsque le gestionnaire est activé. Vous avez toujours la possibilité de repasser l'éditeur en mode 'Arrêt sur toute les erreurs' pour le retrouver plus facilement.

Traitement immédiat

Le traitement immédiat consiste à tester la présence d'une erreur juste après la ligne de code sensée la provoquer. Il s'applique plutôt quand une erreur très prévisible et facile à traiter immédiatement se trouve dans le code, ou quand on cherche à provoquer une erreur comme élément d'une réponse. L'instruction est de la forme :

```
On Error Resume Next
```

Ce qui pourrait se lire comme, en cas d'erreur exécute la ligne suivante. La ligne suivante va donc être un test sur la propriété Number d'ErrObject soit par une égalité si le type de l'erreur attendue est exactement connu, soit juste sous forme de test de valeur non nulle si plusieurs erreurs sont possibles et que le type est indifférent. Généralement on traite donc l'erreur dans le bloc If qui suit la ligne incriminée, on réinitialise l'objet ErrObject et on désactive le gestionnaire pour l'erreur en ligne.

S'il n'y a pas de gestionnaire général, on désactive à l'aide de l'instruction :

```
On Error Goto 0
```

Sinon on réactive le gestionnaire en cours en reprécisant l'étiquette avec :

```
On Error Goto Etiquette
```

Commençons par un exemple hyper classique. Si vous souhaitez enregistrer un classeur avec un nom d'un fichier déjà existant, vous allez obtenir un message d'avertissement d'Excel. Vous pouvez éventuellement supprimer l'affichage de ce message comme nous le verrons plus loin, mais vous pouvez aussi traiter cela par le code.

La solution 'Excel' s'écrirait :

```
Public Sub EcraserFichier1(ByVal NomFichier As String)

    Application.DisplayAlerts = False
    ThisWorkbook.SaveAs NomFichier

End Sub
```

Nous y reviendrons plus en détails dans l'étude de l'objet Application, mais à mon sens cette façon n'est pas toujours une bonne façon de procéder, encore que dans cet exemple précis cela ne pose pas de gros problèmes.

```
Public Sub EcraserFichier2(ByVal NomFichier As String)

    If Len(Dir(NomFichier, vbNormal)) > 0 Then Kill NomFichier
    ThisWorkbook.SaveAs NomFichier

End Sub
```

Cette solution est relativement élégante. Cependant le test sur la fonction Dir peut être faussé selon ce que contient l'argument 'NomFichier' et une erreur pourrait quand même se produire.

```
Public Sub EcraserFichier3(ByVal NomFichier As String)

    On Error Resume Next
    Kill NomFichier
    If Err.Number > 0 Then Err.Clear
    ThisWorkbook.SaveAs NomFichier

End Sub
```

Cette solution utilise le traitement immédiat. Telle qu'elle est écrite, elle exécute le Kill puis réinitialise l'objet ErrObject au cas où Kill ne trouve pas le fichier correspondant. Pour ma part, j'évite ce genre de traitement car nous sommes dans un cas ou d'autres erreurs que l'erreur 53 (Fichier introuvable) peut se produire. Supposons par exemple que l'utilisateur n'est pas les droits d'écriture dans le répertoire cible. L'appel de la fonction Kill va engendrer une erreur de type 70 (Permission refusée). Puisque le test de la ligne suivante sera vrai, l'objet ErrObject sera réinitialisé. Comme l'utilisateur n'aura pas les droits, la ligne demandant l'enregistrement va échouée aussi, mais l'utilisateur ne le saura pas puisque je n'ai pas ramené le gestionnaire d'erreur en mode normal.

Le traitement en ligne peut aussi être utilisé comme élément de réponse. Imaginons que nous souhaitions savoir si un classeur contient une feuille nommée 'Data'. Sans le traitement d'erreurs, nous devrions écrire :

```
Private Function FeuilleExiste(ByVal NomFeuille As String, ByRef Classeur
As Workbook) As Boolean

    Dim EnumFeuille As Worksheet
    For Each EnumFeuille In Classeur.Worksheets
        If StrComp(EnumFeuille, NomFeuille, vbTextCompare) = 0 Then
            FeuilleExiste = True
            Exit Sub
        End If
    Next EnumFeuille

End Function
```

Si le classeur contient 200 feuilles, le traitement aura un coût non négligeable en terme de temps d'exécution. Je peux alors utiliser le traitement d'erreurs et écrire :

```
Private Function FeuilleExiste(ByVal NomFeuille As String, ByRef Classeur
As Workbook) As Boolean

    Dim EnumFeuille As Worksheet
    On Error Resume Next
    Set EnumFeuille = Classeur.Worksheets(NomFeuille)
    If Err.Number = 0 Then FeuilleExiste = True Else Err.Clear

End Fonction
```

Vous trouverez certains 'puristes' qui vous diraient que cette façon de programmer n'est pas propre. Il n'y a aucun argument sérieux à ce purisme là, l'interception d'erreurs lorsqu'elle est bien gérée est une programmation tout à fait correcte.

Erreurs successives

La gestion des erreurs successives est souvent négligée lors de l'écriture du code de gestion d'erreur. Comme il s'agit de mécanismes relativement complexes, on essaye souvent de contourner le problème autant que faire se peut.

Programmation sans échec

Il s'agit généralement d'une faute de programmation. La programmation dite sans échec consiste à activer l'instruction On Error Resume Next sans traiter nécessairement les erreurs éventuelles. Par exemple :

```
Public Sub Sans Echec()

    On Error Resume Next
    Dim compteur As Long
    For compteur = 1 To 10
        MsgBox ThisWorkbook.Worksheets(compteur).Name
    Next compteur
    ThisWorkbook.Worksheets(compteur).Delete

End Sub
```

Ce code ne lèvera jamais d'erreur, affichera autant de fois le message qu'il y a réellement de feuilles et supprimera la onzième feuille si elle existe. Ce type de programmation ne présente pas d'intérêt réel, par contre il masquera des éventuels bugs graves sur des codes un peu plus complexes.

Validation et activation

Le code qui va suivre n'a pas d'autre but que d'expliquer le principe des erreurs successives. Il est particulièrement mal écrit, ne vous en inspirez pas pour écrire des fonctions de recherche.

```
Public Sub Main()

    Call FindDate(Now)
    Selection.Interior.BackColor = vbRed

End Sub

Public Sub FindDate(ByVal UneDate As Date)

    Dim MaCell As Range, Plage As Range
    On Error GoTo Erreur
    For Each MaCell In ActiveSheet.UsedRange.Cells
        If IsDate(MaCell.Value) And DateDiff("d", UneDate, MaCell.Value) >
0 Then
            Set Plage = Application.Union(MaCell, Plage)
        End If
    Next MaCell
    Plage.Select
    Exit Sub
Erreur:
    If Err.Number = 91 Then
        Err.Clear
        ActiveSheet.UsedRange.Find(UneDate).Select
    End If

End Sub
```

Que va-t-il se passer s'il n'y a aucune date postérieure ou égale à la date passée en argument ?

Précisons d'abord quelques termes. Dès lors qu'on entre dans un code couvert par l'instruction On Error Goto Etiquette, on dit que le gestionnaire d'erreurs est validé. Lorsqu'une erreur se produit et que le curseur d'exécution entre dans le code de gestion d'erreurs, on dit que le gestionnaire est actif. Enfin si on repasse par une ligne On Error Goto 0, on dit que le gestionnaire est invalidé.

Donc observons la fonction 'FindDate'. Sa première instruction valide le gestionnaire d'erreurs. Supposons que la feuille ne contienne pas de date postérieure à la date passée en argument. L'appel de l'instruction Plage.Select va lever une erreur 91 puisque Plage vaudra Nothing. Le gestionnaire d'erreurs va donc s'activer. Un gestionnaire actif ne peut pas être de nouveau activé puisqu'on obtiendrait alors une exécution infinie. Dès lors notre code va lever une erreur non interceptée. Modifions notre procédure Main, telle que :

```
Public Sub Main()

    On Error GoTo Erreur
    Call FindDate(Now)
    Selection.Interior.ColorIndex = 3
    Exit Sub
Erreur:
    If Err.Number = 91 Then
        Call MsgBox("Aucune date égale ou postérieure au : " &
DateValue(Now))
    End If

End Sub
```

Pour les mêmes raisons que précédemment, notre fonction 'FindDate' va lever une erreur alors que son gestionnaire est déjà actif. Comme nous l'avons vu, toute erreur récupérable va chercher un gestionnaire validé mais inactif en remontant la pile des appels. Notre nouvelle procédure Main a un gestionnaire qui répond à ces conditions. Il va donc s'activer pour traiter la deuxième erreur de la fonction 'FindDate', ce qui déclenchera l'affichage de la boîte de message.

Ceci implique que si vous devez avoir un code dont le gestionnaire d'erreurs est susceptible de lever une erreur, vous devez placer ce code dans une procédure appelée et écrire un gestionnaire d'erreur dans la procédure appelante.

Modèle objet

Nous allons à partir de maintenant travailler sur le modèle objet d'Excel. Contrairement à ce que l'on croit généralement, cette partie ne présente que peu de difficultés par rapport aux bases du langage que nous avons vu dans la première partie.

Les quelques difficultés de la manipulation de ce modèle viennent de trois sources :

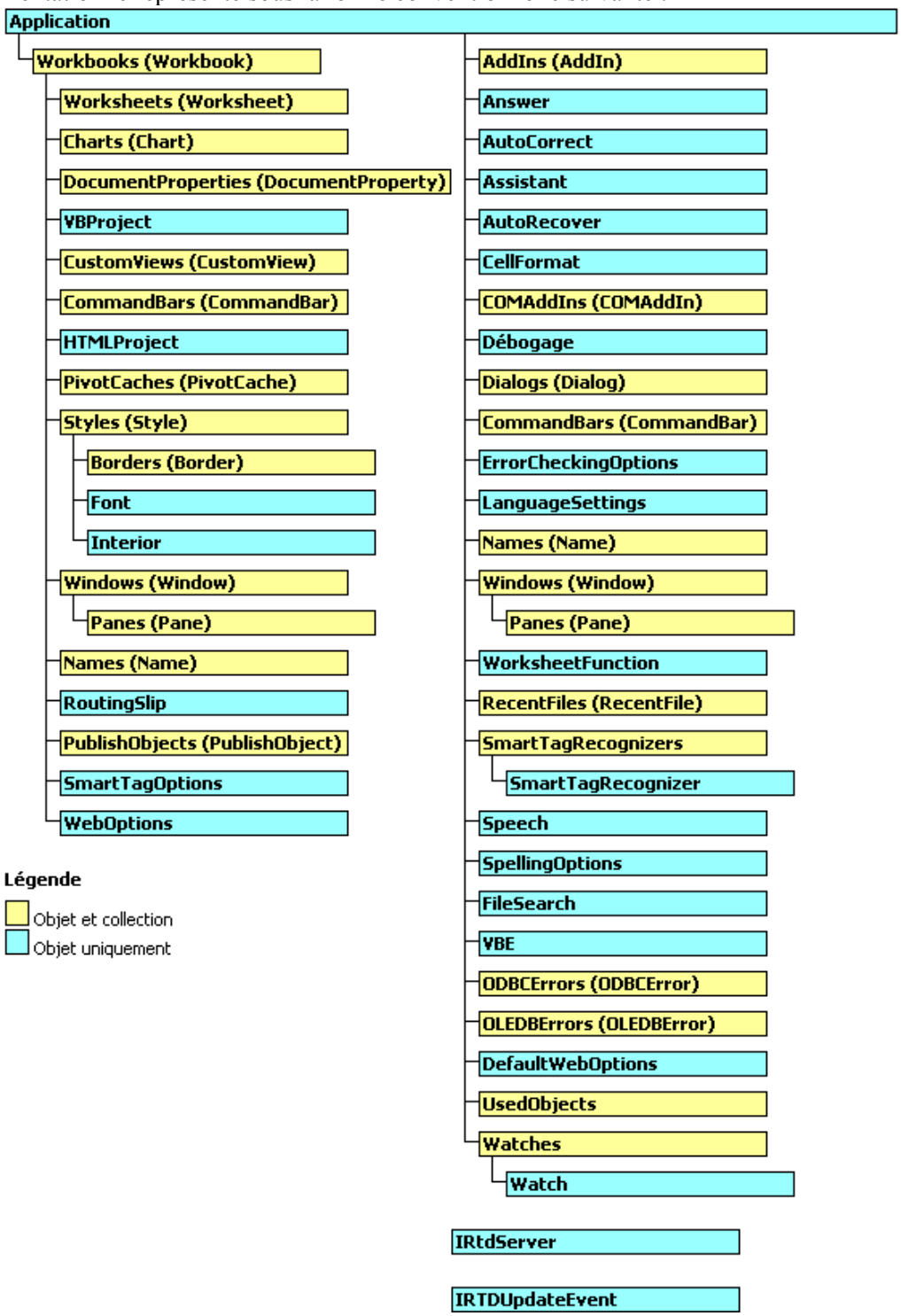
- Une mauvaise compréhension de la manipulation des objets
- Une mauvaise représentation de l'architecture hiérarchique d'Excel
- Quelques ambiguïtés dans le modèle.

Nous allons donc bien détailler ces aspects afin que vous puissiez manipuler le modèle objet Excel comme Geronimo son tomahawk.

Lors de l'étude des objets, la liste des propriétés et méthodes que nous verrons sera limitée à une programmation de base Excel. Sachez qu'il existe dans chaque objet d'autres propriétés et méthodes que nous ne verrons pas dans ce cours mais qui peuvent vous être utiles.

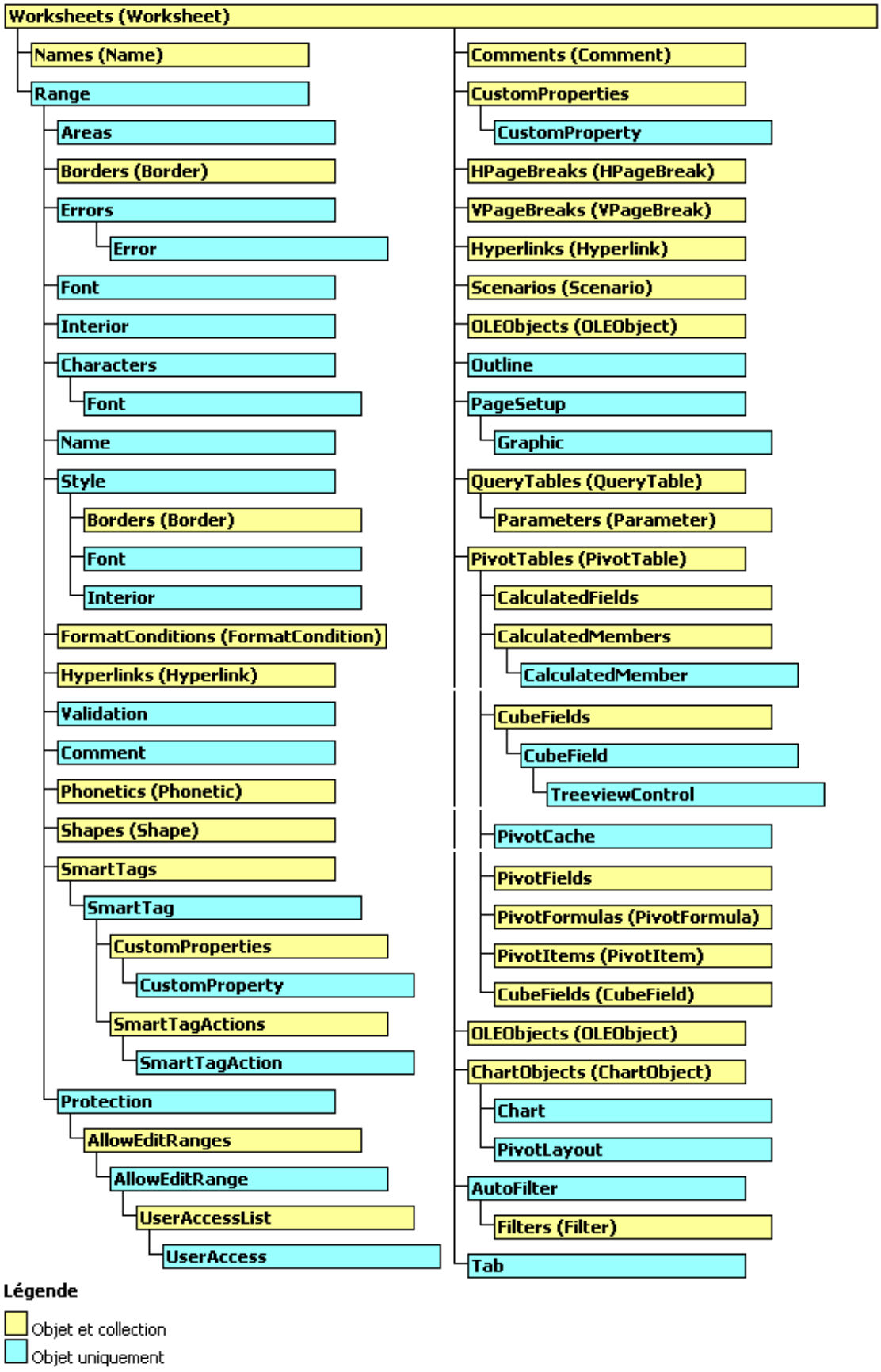
Présentation

Le modèle objet Excel est un monstre énorme lorsqu'on le considère dans son ensemble. La documentation le représente sous la forme conventionnelle suivante :

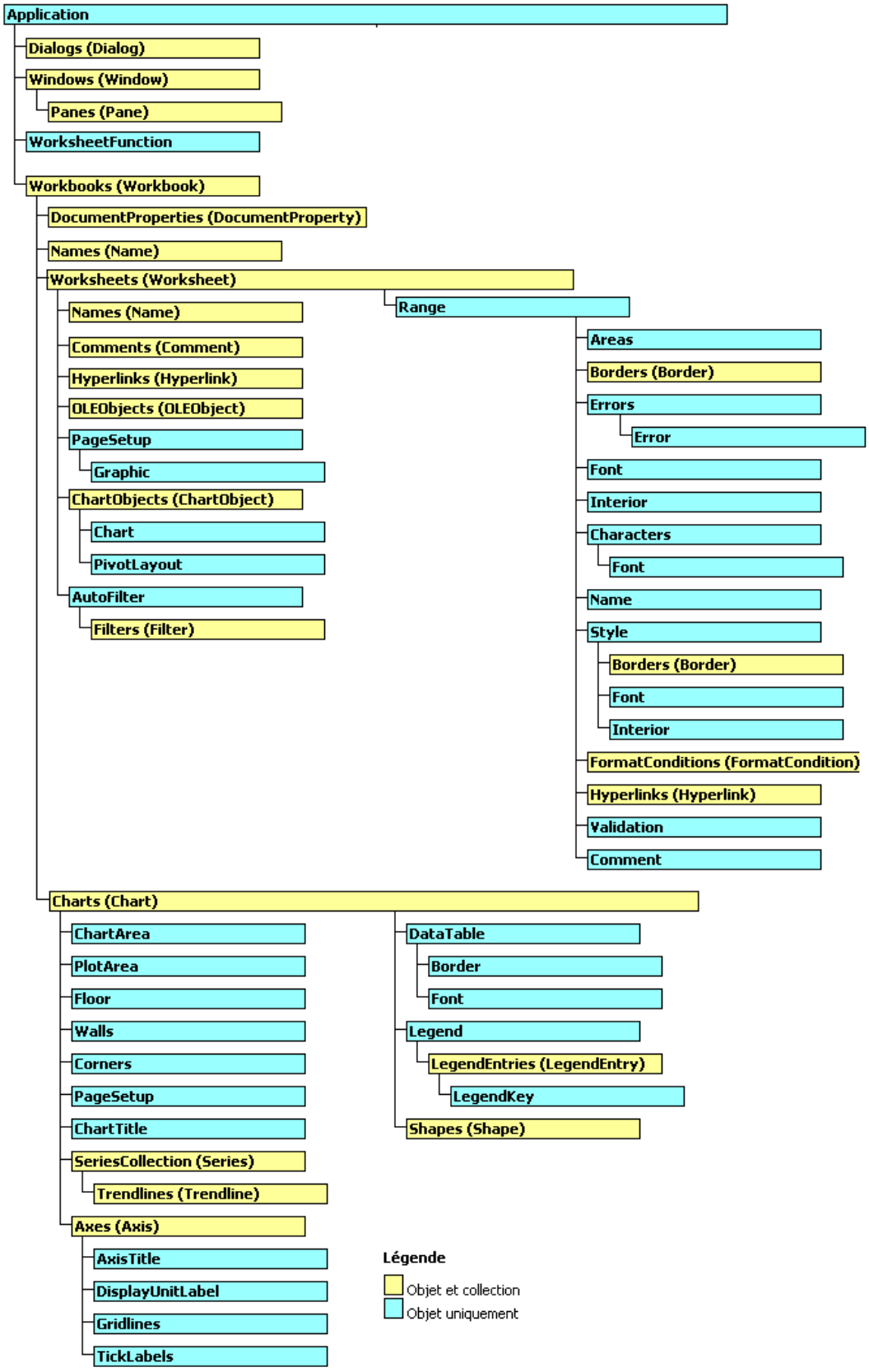


Encore faut il avoir bien à l'esprit qu'il ne s'agit là que d'un premier niveau de hiérarchie.

La simple collection WorkSheets se décompose telle que :



Il n'est pas question ici de détailler l'ensemble de ce modèle objet. Nous allons nous cantonner dans l'étude des principaux objets utilisés dans des macros classiques. Le modèle objet vu dans ce cours sera :



Fondamentaux

Nous allons voir ou revoir dans ce chapitre les fondamentaux de la manipulation des objets en Visual Basic, puis la construction du modèle objet d'Excel et les pièges à éviter.

Glossaire

Je vais d'abord définir quelques termes pour que les explications qui suivent soient plus claires.

Objet : Groupe de code et de données réagissant comme une entité. La plupart des objets que nous allons voir dans le modèle Excel ont une représentation graphique (dans le sens visuel) ce qui les rends plus facile à appréhender. Ainsi, le classeur, la feuille, la (ou les) cellule(s) ou le graphique sont des objets en tant que tels.

Méthode : Dans la théorie objet, tous les membres d'un objet sont des méthodes. Dans le jargon courant, les méthodes sont les fonctions ou les procédures d'un objet influant sur celui-ci. Lorsqu'elles n'influent pas sur lui on les appelle souvent fonctions ce qui ne simplifie pas les choses.

Propriété : Méthode particulière renvoyant ou définissant les champs (les caractéristiques) d'un objet. Les propriétés s'utilisent donc comme les variables, c'est-à-dire qu'on lit ou qu'on affecte leurs valeurs. Certaines propriétés, dites en 'lecture seule' n'autorisent pas l'affectation de valeurs.

Évènement : Un évènement est un 'signal' émis par un objet généralement lors d'une interaction ou d'un changement d'état de celui-ci. Ce signal peut être intercepté à l'extérieur de l'objet par des procédures ayant une signature définie par l'évènement. On appelle alors ces procédures des gestionnaires d'évènements.

Interface : En VBA, c'est l'ensemble des membres publics d'un objet. On entend par membre, les méthodes, propriétés et évènements d'un objet.

Collections : Stricto sensu, une collection est un jeu d'éléments indexés de tous types. Dans le modèle objet d'Excel, les collections renvoient uniquement des objets de même type à l'exception de la collection Sheets.

Qualification : De son vrai nom, chaîne de qualification. En jargon pur sucre, la qualification est l'identification unique de la référence c'est-à-dire le parcours hiérarchique dans le modèle objet pour désigner un objet ou un membre unique.

Les aides dans l'éditeur

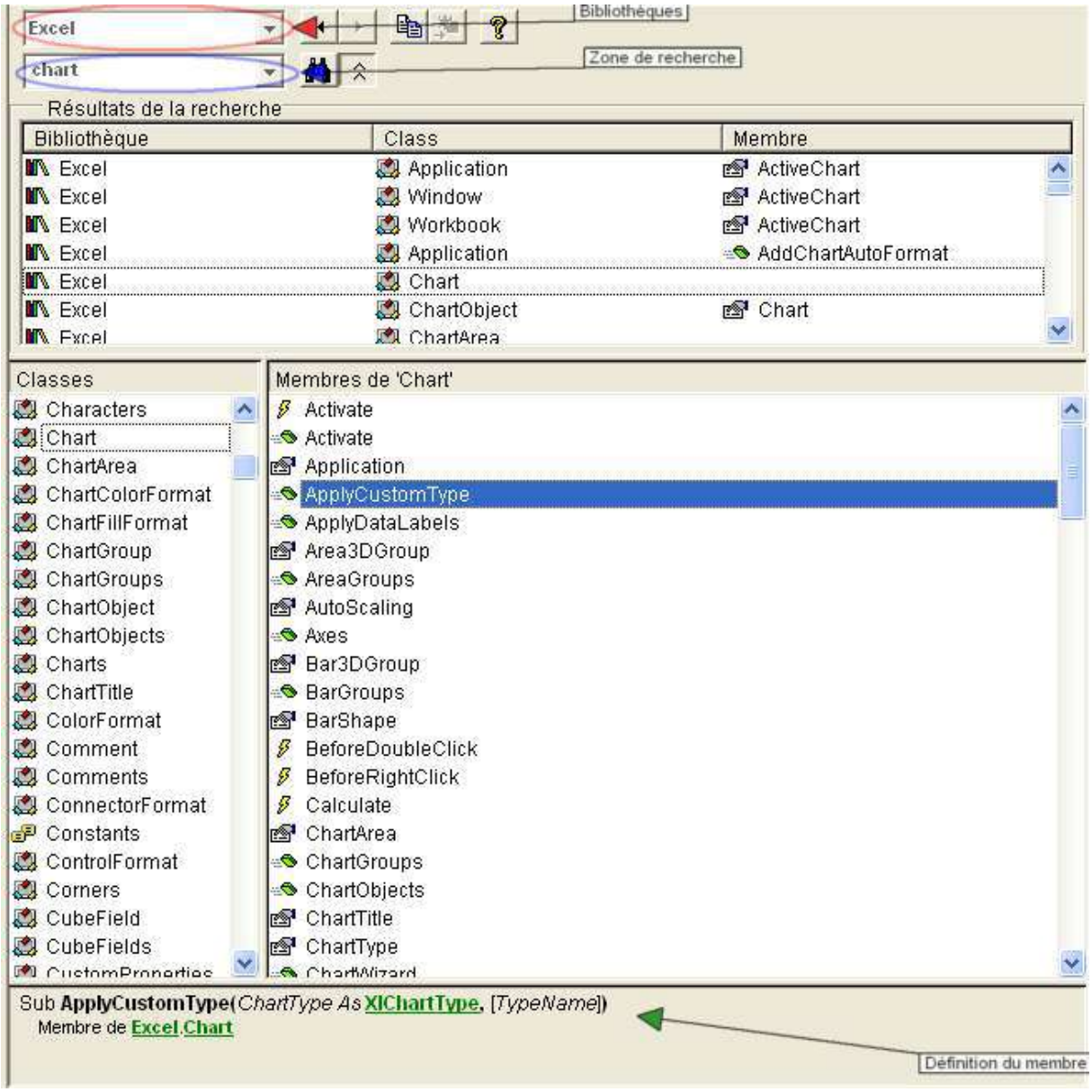
Explorateur d'objet

L'explorateur d'objets remplace le chien en tant que meilleur ami du développeur VBA. Vu la densité du modèle objet, il serait presque impossible d'avoir une vision simple des interfaces de chaque objet. L'explorateur d'objets vous permet cette manipulation. Il est accessible depuis l'éditeur VBA, soit en choisissant le menu "Affichage – Explorateur d'objets", soit en appuyant sur la touche F2.

L'explorateur d'objet utilise les icônes suivantes :

 Bibliothèque
 Module
 Classe
 Énumération
 Type utilisateur
 Méthode
 Méthode par défaut
 Propriété
 Propriété par défaut
 Évènement
 Constante

Dans la définition du membre, les paramètres facultatifs sont entourés de crochets. Lorsque le type est omis il s'agit d'un Variant.

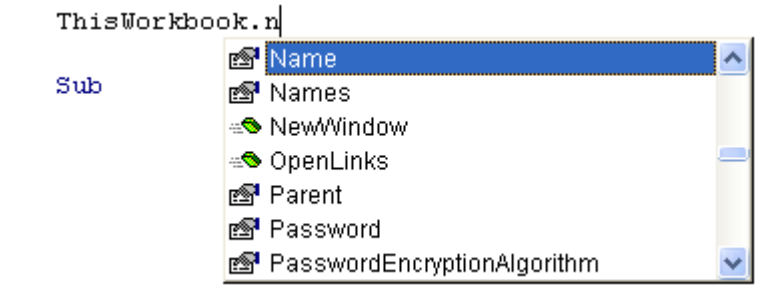


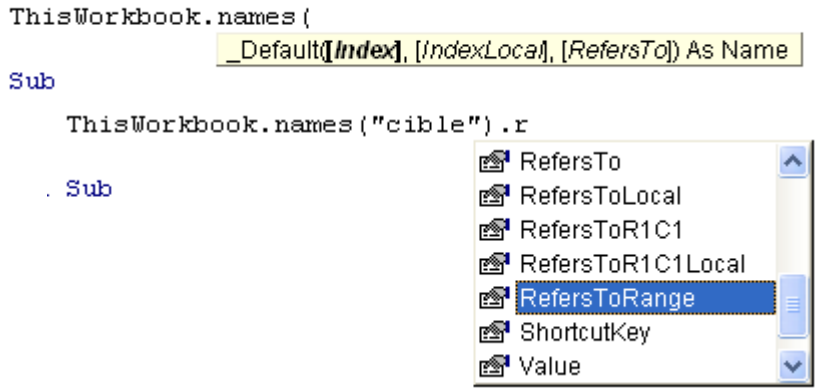
Nous n'allons pas voir ici comment manipuler l'explorateur car c'est assez trivial, je vous invite vivement à vous exercez un peu dessus car une fois qu'on l'a bien pris en main c'est extrêmement pratique. Notez que si vous surlignez un élément dans l'explorateur de d'objet et que vous appuyez sur FI, l'aide correspondante s'affichera.

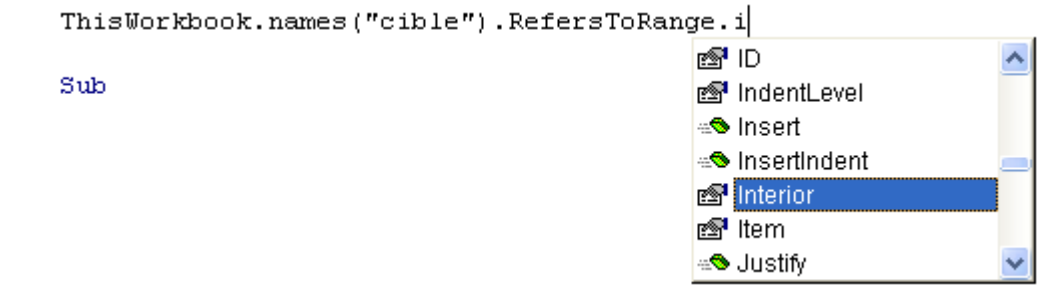
IntelliSense

IntelliSense est un mécanisme d'aide au développement qui se base sur le contexte de ce que vous êtes en train de coder pour vous fournir une aide. La plus connue de ces aides, parce que la plus visible et l'aide à la qualification, plus connue sous le nom de saisie automatique.

Imaginons que vous vouliez changer la couleur de fond de la cellule nommée cible du classeur de la macro. Au fur et à mesure que vous allez parcourir la hiérarchie des objets dans le chemin de qualification, IntelliSense va vous afficher la liste des membres de chaque objet. Cela va donner quelque chose comme :







Et ainsi de suite. Pour sélectionner un élément dans la liste vous pouvez utiliser les touches flèches du clavier ou la souris, pour valider la sélection vous pouvez appuyer sut "Tab" ou double cliquer dessus.

Cette aide semi automatique utilise plusieurs fonctionnalités auxquelles vous pouvez accéder aussi en faisant un clic droit sur un mot, à savoir :

La liste des membres quand il s'agit d'un objet

La liste des paramètres quand il s'agit d'une méthode.

IntelliSense vous permet aussi de compléter un mot en tapant ses premières lettres puis en choisissant le menu contextuel "Compléter un mot".

Manipulation d'objets

Bien que nous en ayons déjà parlé précédemment, nous allons revoir en détail la manipulation des objets.

Durée de vie & Portée

Je ne vais pas entrer dans les détails du fonctionnement des objets. Disons pour faire simple qu'un objet doit toujours être créé (en jargon on dit instancié) et détruit (là on dit finalisé). Le moment entre ses deux évènements est la durée de vie d'un objet. Vous ne pouvez utiliser un objet que pendant sa durée de vie et uniquement dans sa portée c'est-à-dire d'un endroit où il est visible selon qu'il soit Public ou Privé.

On crée un objet de deux manières en VBA :

1) A l'aide du mot clé New, soit dans la déclaration de la variable, soit dans une instruction de code. Les deux méthodes ne sont pas tout à fait équivalentes et on tend plutôt à instancier dans une instruction spécifique. Autrement dit :

```
Sub TestObjet()

    Dim MaCollection As Collection

    Set MaCollection = New Collection
    MaCollection.Add "élément1"
    MaCollection.Add "élément2"

End Sub
```

Vaut mieux que

```
Sub TestObjet()

    Dim MaCollection As New Collection

    MaCollection.Add "élément1"
    MaCollection.Add "élément2"

End Sub
```

2) A l'aide de la fonction CreateObject.

Celle-ci suit le modèle suivant :

Function CreateObject(*Class As String*, [*ServerName As String*])

Où l'argument *Class* est formé par le couple *appname.objecttype* où *appname* est le nom de l'application qui crée l'objet et *objecttype* le type de l'objet créé. Par exemple :

```
Dim rs As Object

    Set rs = CreateObject("ADODB.Recordset")
```

Dans l'absolu, Il n'y a pas de différence à utiliser New ou CreateObject. Cependant dans le contexte de la programmation il peut y avoir un risque.

En effet, vous ne devez instancier que des objets que votre code crée directement. La plupart des objets que vous allez manipuler dans le modèle objet d'Excel **sont créés par Excel** et vous ne devez pas les instancier. Pourquoi cela ?

Lorsque vous manipulez Excel par du code VBA, il existe forcément une instance de l'application Excel en cours et un classeur en cours, celui qui contient votre code. Si à un quelconque moment vous instanciez l'objet application, vous allez créer **une deuxième instance** d'Excel. Pour vous en convaincre, il suffit d'exécuter le code suivant :

```
Sub TestObjet()

Dim xlApp As Excel.Application

    'récupère l'instance en cours d'Excel
    Set xlApp = Application
    'lance une deuxième fois Excel
    Set xlApp = New Application
    xlApp.Visible = True

End Sub
```

Comme Excel suit un modèle hiérarchique, il ne va pas vous laissez instancier un des objets enfants de l'objet application, puisque par exemple il ne peut pas exister une feuille qui ne soit pas membre d'un classeur. Ainsi pour créer une feuille, on utilise une méthode de l'objet Workbook telle que :

```
Dim Feuille As Worksheet
    Set Feuille = ThisWorkbook.Sheets.Add()
```

Ce code ajoute bien une nouvelle feuille au classeur, mais c'est Excel qui c'est chargé d'instancier l'objet. Si vous essayez de faire :

```
Set Feuille = New Worksheet
```

Vous allez lever une erreur, car cette feuille ne peut pas être créée en dehors d'un classeur.

Cependant on pourrait faire :

```
Sub TestObjet()

Dim Feuille As Object

    Set Feuille = CreateObject("Excel.Sheet")
    Feuille.Application.Visible = True

End Sub
```

Et là il n'y aura pas d'erreur, mais vous allez créer un nouveau classeur contenant une feuille et non simplement une feuille. L'explication du pourquoi est un peu technique, donc il faut surtout vous rappeler de privilégier l'utilisation du mot clé New.

Pour détruire un objet, il suffit de le mettre à Nothing. Vous ne pouvez pas détruire un objet qu'Excel a instancié, mais vous pouvez réinitialiser une variable objet pointant sur un de ces objets.

```
Sub TestObjet()

Dim xlApp As Excel.Application, MaColl As Collection

    Set MaColl = New Collection
    MaColl.Add "élément1"
    'détruit l'objet collection
    Set MaColl = Nothing
    Set xlApp = Application
    'réinitialise la variable xlApp
    Set xlApp = Nothing
    'écriture interdite, impossible de détruire une instance créé par Excel
    'Set Application = Nothing

End Sub
```

Cet aspect des choses est un peu complexe, mais dans le modèle objet Excel et en utilisant VBA, vous n'aurez quasiment jamais à instancier les objets Excel, il existe presque toujours une méthode Add où Excel prends le mécanisme en charge. Autrement dit, vous n'avez pas à gérer l'instanciation ni la finalisation des objets Excel.

Qualification & Manipulation des membres

La qualification, c'est donc le chemin exact d'un membre spécifique d'un objet défini. La première erreur à ne pas commettre est de confondre le qualificateur et le type. Par exemple si nous écrivons :

```
    Dim Police As Excel.Font
    Set Police = ThisWorkbook.Worksheets(1).Range("A1").Font
```

Le type de la variable 'Police' est Excel.Font (ou Font car Excel peut être omis dans la déclaration) alors que le qualificateur, c'est-à-dire la désignation d'un objet Font défini est ThisWorkbook.Worksheets(1).Range("A1").

On utilise l'opérateur . (point) pour la qualification, sous la forme Objet.Membre. Évidemment, si le membre renvoie un objet, on peut écrire Objet.Membre.Membre et ainsi de suite. Généralement, l'IntelliSense vous fournit les éléments de l'interface de l'objet en cours mais vous constaterez que dans certains cas, alors que le membre en cours renvoie bien un objet, il n'y pas d'affichage de l'IntelliSense.

Par exemple si vous tapez ActiveWorkbook. IntelliSense va afficher les membres de l'objet Workbook, par contre, si vous tapez ActiveSheet. Rien ne va s'afficher. Il ne s'agit pas d'un bug mais d'un phénomène dû à la définition de la méthode. Pour comprendre cela, activer l'explorateur d'objets en appuyant sur F2, sélectionnez la classe Application puis le membre ActiveWorkbook. La définition du membre est :

Property **ActiveWorkbook** As **Workbook**

Sélectionnez maintenant le membre ActiveSheet, et vous verrez la définition :

Property **ActiveSheet** As Object

Comme le type renvoyé est Object et que ce type n'a pas de membres, IntelliSense n'affiche rien. Ce phénomène de rupture est assez fréquent lorsqu'on renvoie un membre d'une collection. Si vous connaissez le type sous-jacent renvoyé, vous pouvez utiliser une variable objet typée correctement pour rétablir l'IntelliSense. Par exemple :

```
Sub TestQualif()
    Dim Police As Font, Feuille As Worksheet
    Set Feuille = ActiveSheet
    Set Police = Feuille.Range("A1").Font
End Sub
```

Attention toutefois, ce code est potentiellement source d'erreurs. Si la propriété ActiveSheet renvoie une référence de type Object, c'est parce que la feuille active n'est pas forcément une feuille de calcul mais peut être aussi une feuille graphique ou une feuille macro.

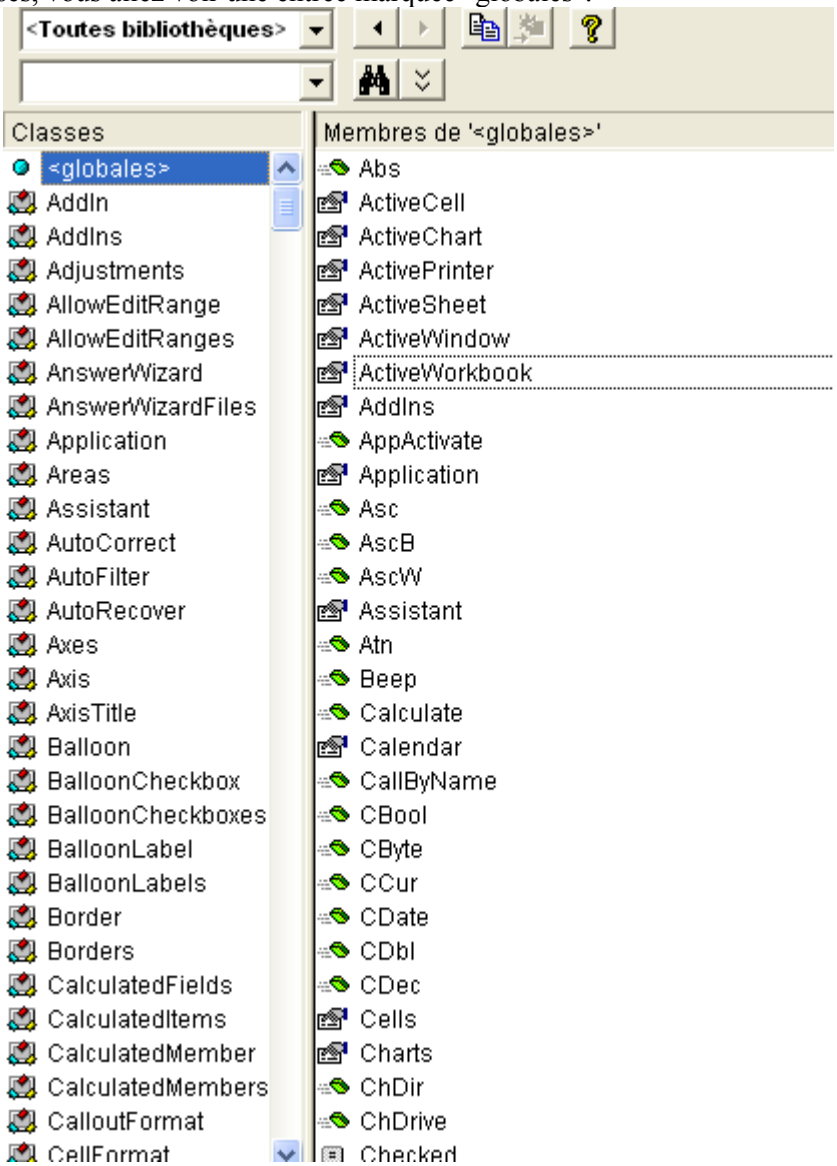
Tous les membres ne renvoient pas des objets, n'oubliez pas que les propriétés se gèrent comme des variables, c'est-à-dire qu'on récupère ou affecte des valeurs, alors que les méthodes se gèrent comme des fonctions. Certaines méthodes renvoient un résultat, dans ce cas la règle des parenthèses s'applique avec ou sans l'utilisation de Call.

```
Sub TestQualif()
    Dim Police As Font, Couleur As Long, Valeur As String
    Dim Commentaire As Comment, Plage As Range

    Valeur = "Test"
    'affecte un objet Range à la variable objet
    Set Plage = Range("A1")
    'affecte un objet Font à la variable objet
    Set Police = Plage.Font
    'Lit la valeur d'une propriété
    Couleur = Plage.Interior.Color
    'Affecte une valeur à une propriété
    Plage.Value = Valeur
    'Exécute une méthode et récupère la référence renvoyée
    Set Commentaire = Plage.AddComment("commentaire")
    'Exécute une méthode et ne récupère pas la valeur renvoyée
    Call Commentaire.Text("c'est un ", 1, False)
    's'écrit aussi
    'Commentaire.Text "c'est un ", 1, False
End Sub
```

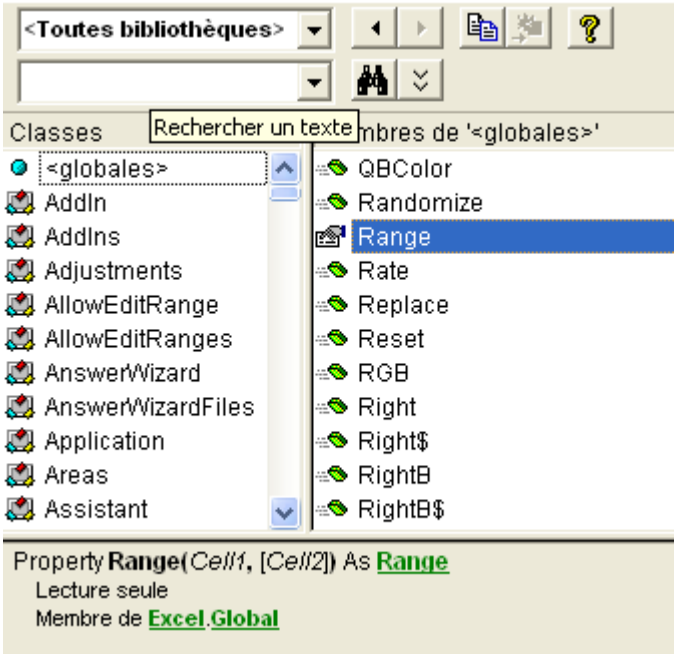
Ce code exemple fonctionnera sans problème. Cependant si on regarde la deuxième instruction

```
Set Plage = Range("A1")
```

On voit bien que l'objet Range n'est pas qualifié. Comment expliquer alors que cela fonctionne. En fait, les langages Visual Basic utilisent un principe, appelé Scope, qui permet d'utiliser certains objets ou fonctions sans qualification. Le principe de fonctionnement du scope est assez complexe à appréhender, aussi n'allons nous pas l'étudier ici. Cependant vous pouvez le visualiser facilement dans l'explorateur d'objets. Si vous appuyez sur F2, et que vous regardez l'affichage par défaut du volet des classes, vous allez voir une entrée marquée "globales".



Les membres de cette entrée sont les membres du scope VBA, c'est-à-dire les membres qui n'ont pas besoin d'être qualifiés pour être utilisés. Si nous cherchons le membre Range, nous trouvons :



Comme vous le voyez, cette propriété est en lecture seule, c'est-à-dire que vous ne pouvez pas y affecter un objet Range. Un objet Range représente une ou plusieurs cellules comme nous le verrons bientôt en détail. En soit, cela n'a pas d'existence au niveau de l'application, puisque une plage de cellules appartient forcément à une feuille de calcul. En fait, Excel utilise le concept d'objets actifs ou sélectionnés. Ainsi si vous regardez les premiers membres définis dans 'globales', vous trouverez le classeur actif, la feuille active, la fenêtre active, etc.

L'objet Range que vous pouvez atteindre sans qualification correspond donc à une plage de cellule, de la feuille active du classeur actif. Cette notion étant pour le moins dangereuse, il convient de bien comprendre ce qu'est une référence.

Gérer les références

Dans la programmation Excel, on travaille sur des objets existants. La qualification ou l'utilisation de variable objet n'a pas d'autre but que vous permettre de désigner spécifiquement un objet. Peu vous importe de savoir où et comment est gérer cet objet en réalité, il suffit que vous sachiez le désigner. Pour pouvoir faire cela, vous allez manipuler des références à cet objet qui ne sont rien d'autres que des représentations de son adresse.

Excel maintient un certain nombre de références temporaires, telle que le classeur actif ou la feuille active, seulement celles-ci peuvent changer au cours du temps sous l'action d'Excel, de votre code ou de l'utilisateur. C'est pour cela qu'il faut généralement ne pas les utiliser sauf si vous avez **la certitude** que pendant le temps de votre exécution, elles désigneront forcément les bons objets.

Ces références temporaires dans Excel s'appellent :

- ActiveCell
- ActiveChart
- ActiveSheet
- ActiveWindow
- ActiveWorkbook
- Selection

Les membres suivants lorsqu'ils sont utilisés sans qualificateur renvoient à l'objet actif et présentent donc les mêmes inconvénients.

- Cells
- Columns
- Names
- Range
- Rows

Vous ne devez pas perdre de vue que ces éléments n'existent pas obligatoirement, si la feuille active est une feuille graphique par exemple, l'appel d'un des membres de la liste ci-dessus va lever une erreur puisqu'ils n'ont d'existence que si ActiveSheet est une feuille de calcul.

Par contre il existe deux références fiables, Application qui pointe vers la session Excel dans laquelle à été ouvert le classeur contenant votre code, et ThisWorkbook qui désigne le classeur contenant votre code, puisque celui-ci est forcément ouvert pour que le code puisse s'exécuter.

Si vous définissez une variable objet sur une de ces références temporaires, votre référence elle deviendra fixe. Autrement dit, la modification de la référence temporaire ne modifiera pas la référence de votre objet.

```
Dim MaFeuille As Worksheet

If Not ActiveSheet Is Nothing Then
    Set MaFeuille = ActiveSheet
    MaFeuille.Next.Activate
    MsgBox MaFeuille.Name & vbNewLine & ActiveSheet.Name
End If
```

Nous allons obtenir l'affichage de :

Au début du code, ActiveSheet fait référence à la feuille 'Feuil1' du classeur actif. J'affecte cette référence à la variable *'MaFeuille'*, qui du coup, pointe elle aussi sur la même feuille. J'active la feuille suivante du classeur à l'aide de l'instruction :

```
    MaFeuille.Next.Activate
```

ActiveSheet fait alors référence à la nouvelle feuille active, c'est-à-dire 'Feuil2'. Cependant ma variable *'MaFeuille'* n'est pas modifiée par le changement de feuille active, dès lors on a un affichage des deux noms différents puisque les deux références sont différentes.

L'opérateur Is : Cet opérateur définit si deux désignations font référence à un même objet. L'expression Obje x si non.

Architecture Excel

Tout cela peut encore paraître un peu obscur si vous ne faites pas l'analogie entre le modèle objet Excel et l'application Excel telle qu'elle est architecturée. Lorsque vous lancez Excel, vous avez la possibilité de manipuler plusieurs classeurs, chacun pouvant contenir plusieurs feuilles de types identiques ou différents (feuille de calcul, graphique), chaque feuille de calcul contenant 16777216 cellules rangées sous la forme 65536 lignes et 256 colonnes. Vous pouvez manipuler ces cellules une par une ou sous la forme de plage continue ou discontinue.

Le modèle objet Excel fonctionne exactement de la même façon. Vous n'avez qu'un seul objet Parent, Application. Celui-ci va vous renvoyer par le biais de propriétés des objets ou des collections d'objets connexes qu'il contient. Ainsi par sa propriété Workbooks il vous donnera accès à tous les classeurs ouverts, les objets Workbook, chacun renvoyant toutes ses feuilles par sa collection Sheets. Les feuilles de calculs exposent leurs cellules par le biais de propriétés Cells, Rows, Columns, mais aussi le travail sous forme de plage par l'objet Range. Un objet Cells expose aussi des objets Borders pour l'encadrement, un Objet Interior pour le corps de la cellule, etc.

Les graphiques seront des objets Charts lorsqu'ils sont sous forme de feuilles graphiques et leur parent sera alors un classeur (Workbook) ou des objets ChartObjects lorsqu'ils seront sous la forme de graphiques incorporés à une feuille de calcul et leur parent sera alors un objet Worksheet.

Il n'y a jamais dissociation de l'architecture d'Excel et du modèle objet manipulé, si vous avez clairement à l'esprit qui contient quoi dans Excel, vous connaissez le modèle objet Excel.

Dès lors, qualifier un objet devient d'une simplicité biblique. Si je veux encadrer la plage de cellules A1:C100 de la feuille de calcul nommée 'Données' du classeur ouvert Calcul.xls, il me suffira d'écrire :

```
Application.Workbooks("calcul.xls").Worksheets("Données").Range("A1:C100").
BorderAround xlContinuous
```

Évidemment, si le code est amené à manipuler plusieurs fois ce classeur, ou cette feuille ou cette plage, il est plus simple de déclarer des variables objets adéquates car le code sera plus facile à écrire.

Ainsi le code précédant pourrait être de la forme :

```
Dim MaFeuille As Worksheet

Set MaFeuille = Application.Workbooks("calcul.xls").Worksheets("Données")
MaFeuille.Range("A1:C100").BorderAround xlContinuous
```

Je pourrais ensuite utiliser partout la variable MaFeuille en lieu et place du qualificateur Application.Workbooks("calcul.xls").Worksheets("Données").

Il est bien question ici de l'architecture Excel et non du mode de fonctionnement. Dans son mode utilisateur, qui est aussi celui de l'enregistreur de macro par la force des choses, Excel agit en mode sélection- Action. Le même code aurait alors une forme similaire à :

```
    Workbooks("calcul.xls").Activate
    Sheets("Données").Select
    Range("A1:C100").Select
    Selection.BorderAround xlContinuous
```

C'est justement cette forme qu'il faut s'appliquer à ne pas reproduire car elle produit un code lent et peu lisible.

Le deuxième aspect architectural à appréhender et la manipulation des cellules et donc le système de référence des feuilles de calcul.

Excel utilise indifféremment deux modes de notation et deux types de références. Les modes de notation dépendent de la façon dont sont identifiées les colonnes. La mode "A1" les identifie à l'aide de lettres, le mode "L1C1" (R1C1" en anglais) par des nombres. Les références sont absolues ou relatives. Une référence absolue renvoie une position de cellule en fonction de sa position dans la feuille, une référence relative renvoie une position en fonction d'une autre position. Une adresse peut être aussi partiellement relative en ligne ou en colonne. 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(1)

En VBA, les notions relatives et absolues existent aussi dans la façon de programmer comme nous le verrons dans l'étude de l'objet Range. Par contre je vous conseille vivement de choisir un type de notation et de vous y tenir, sinon le code est rapidement assez pénible à lire.

Les pièges

Les pièges du modèle objet Excel ne sont pas très nombreux, il est indispensable de bien les connaître.

Référence implicite

Le premier est la possibilité à cause du scope de créer des références implicites pouvant lever des erreurs. Prenons un code exemple.

```
Sub ImplicitRef()

Dim MaFeuille As Worksheet

    Set MaFeuille = ThisWorkbook.Worksheets("Feuil1")
    MaFeuille.Next.Activate
    MaFeuille.Range(Cells(1, 1), Cells(10, 5)).ClearContents

End Sub
```

La notation Range(Cells(),Cells) renvoie une plage de cellule continue entre les deux cellules définies comme argument. Autrement dit :

```
    Range(Cells(1, 1), Cells(10, 5))
```

Renvoie la plage "A1:E10". Pourtant mon code exemple va lever une erreur 1004 indiquant l'impossibilité de résoudre la méthode Range. Pourquoi cela ?

Nous devons nous rappeler de l'existence du scope. Si la propriété Range utilisée est correctement qualifiée, les deux propriétés Cells ne le sont pas. Comme le Scope renvoie des propriétés Cells, ce sont elles qui sont prises en compte. Celles-ci sont des références à la feuille active, alors que Range fait référence à une autre feuille. Il n'est évidemment pas possible de construire une référence sur une feuille en utilisant des références d'une autre feuille. Les propriétés Cells doivent donc être qualifiées à l'identique ce qui se notera :

```
    MaFeuille.Range(MaFeuille.Cells(1, 1), MaFeuille.Cells(10,
5)).ClearContents
```

Ou alors

```
    With MaFeuille
        .Range(.Cells(1, 1), .Cells(10, 5)).ClearContents
    End With
```

La propriété Sheets

Les objets Workbook (classeur) exposent une propriété Sheets qui renvoie une collection. Si la plupart des collections d'objets du modèle Excel renvoient des objets de même type, ce n'est pas le cas de celle-ci. Les feuilles d'un classeur peuvent être de plusieurs types différents, feuilles de calcul, graphiques, macros. Vous devez donc faire attention en utilisant cette collection et privilégier plutôt les collections Charts ou Worksheets.

Membre par défaut

De nombreux objets possèdent un membre par défaut, généralement une propriété. Si votre code est ambigu, l'interpréteur peut croire que vous faites référence à cette propriété par défaut plutôt qu'à l'objet. Regardons le code suivant :

```
Sub TestDefaut()

Dim MaCellule As Variant

MaCellule = ThisWorkbook.Worksheets(1).Range("A1")
Debug.Print MaCellule & " " & TypeName(MaCellule)
'21 Double

End Sub
```

La variable *MaCellule* contient la valeur de la cellule A1. En effet, pour l'interpréteur, si j'avais voulu récupérer la référence de l'objet cellule, j'aurais écris :

```
Set MaCellule = ThisWorkbook.Worksheets(1).Range("A1")
```

Comme je n'ai pas mis Set, il lit la ligne comme étant

```
MaCellule = ThisWorkbook.Worksheets(1).Range("A1").Value
```

Puisque Value est la propriété par défaut de l'objet Range.

Application

L'objet Application représente donc une instance d'Excel.

Propriétés renvoyant des collections

Les propriétés Cells, Charts, Columns, Rows, Sheets, Names etc., sont des références implicites au niveau de l'objet Application. Nous allons justement nous habituer à ne pas les utiliser.

CommandBars

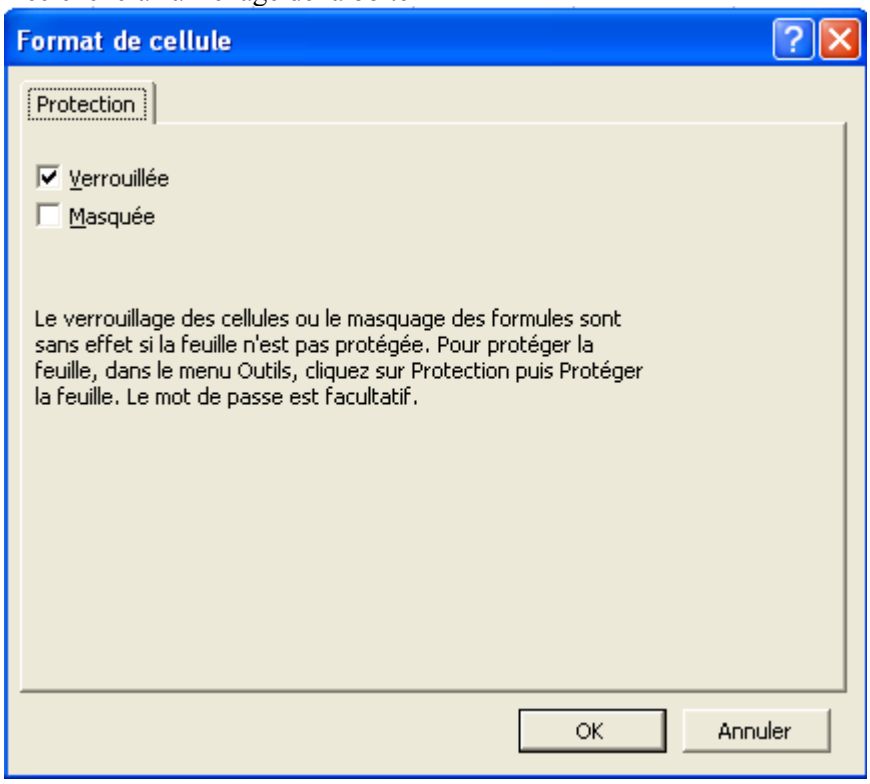
Renvoie la collection des menus et des barres d'outils de l'application Excel. Nous ne traiterons pas ici de la manipulation des barres de menus..

Dialogs

Renvoie la collection Dialogs des boîtes de dialogues prédéfinies dans Excel. Par exemple l'instruction de la ligne :

```
Application.Dialogs(xlDialogCellProtection).Show True, False
```

Déclenchera l'affichage de la boîte



L'utilisation correcte des boîtes de dialogues intégrées n'est pas triviale, nous ne nous l'étudierons pas pour l'instant.

Windows

Renvoie la collection de toutes les fenêtres de l'application. On manipule rarement les fenêtres dans le code VBA puisque l'on cherche souvent à coder sans interaction visuelle. On utilise plus généralement la collection Windows renvoyée par l'objet Workbook.

Workbooks

Renvoie la collection des classeurs ouverts dans la session Excel. Cette collection permet aussi d'ouvrir ou d'ajouter des classeurs.

Propriétés

Les propriétés de l'objet Application influent sur la configuration d'Excel. Par convention, vous devez remettre ces propriétés à la valeur d'origine avant la fin de la session.

Calculation & CalculateBeforeSave (Boolean)

La propriété Calculation définit le mode de calcul d'Excel. Elle peut prendre les valeurs :

xlCalculationAutomatic

xlCalculationSemiautomatic

xlCalculationManual

On peut avoir intérêt à bloquer le calcul automatique lorsqu'on modifie des cellules dans des classeurs contenant beaucoup de formule si on n'a pas besoin de résultats immédiatement dans le code.

La propriété CalculateBeforeSave définit si le calcul doit être forcé pour les classeurs au moment de leur sauvegarde.

Caller

Renvoie un identificateur pour l'appelant. On utilise généralement cette propriété pour savoir d'où vient l'appel d'une fonction VBA. Application.Caller renvoie un objet Range si l'appel vient d'une cellule, une chaîne ou un identificateur pour des macros prédéfinies ou une erreur dans les autres cas.

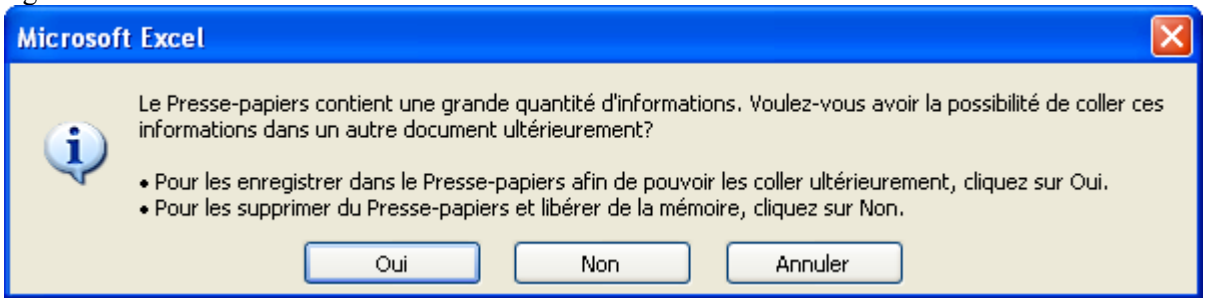
```
Public Function Conv2Farenheit(ByVal TempCelsius As Double) As Double

    Dim Reponse As Double
    Reponse = 1.8 * TempCelsius + 32
    If TypeOf Application.Caller Is Range Then
        Conv2Farenheit = Reponse
    Else
        MsgBox Reponse & " °F"
    End If

End Fonction
```

Le code suivant renvoie la conversion en Fahrenheit d'une température comme valeur de la cellule si une cellule est l'appelant ou sous forme de message sinon.

La locution **TypeOf** *Object* **Is Type** permet de comparer le type de l'objet à un type existant. Elle renvoie vrai si le type d'objet est le même que le type passé.

CutCopyMode (Boolean)

Bien qu'en apparence elle ne serve pas à grand-chose, mettre cette propriété à False permet d'annuler le mode copie en cours avant la fermeture du classeur ce qui supprime l'affichage du message d'alerte



Le code est généralement de la forme :

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    If Application.CutCopyMode = xlCopy Or Application.CutCopyMode = xlCut
Then
        Application.CutCopyMode = False
    End If
End Sub
```

DecimalSeparator (String)

Permet de préciser le séparateur décimal. Ceci évite de recevoir des erreurs de type lorsqu'on travaille avec des fichiers utilisant le séparateur international.

On peut donc écrire un code tel que :

```
    If Application.International(xlDecimalSeparator) = "," Then
        With Application
            .DecimalSeparator = "."
            .ThousandsSeparator = " "
            .UseSystemSeparators = False
        End With
        ChangeSep = True
    End If
… Traitement
    'rétablissement du séparateur d'origine
    If ChangeSep Then
        Application.DecimalSeparator = ","
    End If
```

DisplayAlerts (Boolean)

Supprime les messages d'avertissement d'Excel et applique l'option par défaut lorsque DisplayAlerts vaut False. Vous n'avez pas besoin de rétablir la propriété en fin de code, Excel la ramène systématiquement à la valeur Vrai en fin d'exécution.

Le code suivant demande un nom de fichier à l'utilisateur et sauvegarde le classeur contenant la macro sous se nom en écrasant l'ancien fichier sans avertissement le cas échéant.

```
Dim Fichier As String

    Application.DisplayAlerts = False
    Fichier = Application.GetSaveAsFilename
    If Fichier <> False Then ThisWorkbook.SaveAs Fichier
```

EnableCancelKey (XlEnableCancelKey)

Permet de définir le comportement des touches d'interruption de l'exécution du code.

Peut prendre les valeurs

xlDisabled

xlErrorHandler

xlInterrupt

Le mode *Interrupt* laisse la possibilité d'interrompre le code avec ESCAPE ou CTRL+PAUSE, le mode *ErrorHandled* lève l'erreur 18 (qui peut être interceptée) en cas d'appui sur les touches, le mode *Disabled* désactive les touches. Attention, en cas de boucle infini, il n'est plus possible d'interrompre le code en mode *Disabled*.

EnableEvents (Boolean)

Permet de bloquer les évènements et donc de ne pas exécuter le code évènementiel des objets Excel. Il convient de toujours remettre cette variable à Vrai en fin d'exécution, sinon plus aucun évènement ne sera pris en compte dans la session Excel.

Interactive (Booléen)

Permet d'autoriser ou d'interdire les interactions entre les périphériques de saisie et Excel, à l'exception des boîtes de dialogue.

International

De la forme Application.International(Index) où index est une valeur prédéfinie permettant de préciser le paramètre international que l'on souhaite récupérer. Cette propriété est en lecture seule, elle ne peut pas servir à modifier un ou plusieurs paramètres de la machine.

Il existe de nombreuses valeurs possibles pour index, qu'on retrouve dans l'énumération XlApplicationInternational (et dans l'aide en ligne).

Les plus fréquemment utilisées sont :

Co	Valeur	Type renvoy	Commen
XICountrySetting	2	**Long**	Paramètre de pays/région sélectionné dans le Panneau de configuration Windows.
xICurrencyBefore	37	**Boolean**	**True** si le symbole monétaire précède la valeur, **False** s'il suit la valeur.
xICurrencyCode	25	**String**	Symbole monétaire.
xICurrencyDigits	27	**Long**	Nombre de décimales à utiliser dans les formats monétaires.
xI24HourClock	33	**Boolean**	**True** pour le format 24 heures, **False** pour le format 12 heures
xI4DigitYears	43	**Boolean**	**True** si les années comportent quatre chiffres, **False** si elles comportent deux chiffres.
xIDateOrder	32	**Long**	Ordre des éléments de la date : 0 = mois-jour-année 1 = jour-mois-année 2 = année-mois-jour
xIDateSeparator	17	**String**	Séparateur de date (**/**).
xIMetric	35	**Boolean**	**True** pour le système métrique, **False** pour le système de mesure anglais.
XIDecimalSeparator	3	**String**	Séparateur décimal.
XIThousandsSeparator	4	**String**	Zéro ou séparateur de milliers

Je ne vais pas vous montrer ici d'exemple complet de programmation international, surtout qu'il est à mon sens inutile d'utiliser une programmation adaptée en anglais, Excel se chargeant dans la plupart des cas de localiser correctement.

ScreenUpdating (Boolean)

Désactive la mise à jour d'écran quand elle vaut False. Vous devez la rétablir en fin de procédure. Désactiver accélère le code de façon très importante. Cette accélération est vraie même si votre code n'a pas d'action visible à l'écran. On l'utilise généralement systématiquement pour les procédures longues ou si vous avez de nombreuses mises à jour d'écran, sauf à vouloir rendre les utilisateurs épileptiques.

SheetsInNewWorkbook (Long)

Définit le nombre de feuilles de calcul contenu par défaut dans les nouveaux classeurs. Cela permet de créer des classeurs spécifiques sans avoir à manipuler les feuilles.

```
NbFeuilleDefaut = Application.SheetsInNewWorkbook
Application.SheetsInNewWorkbook = 5
Application.Workbooks.Add 'ce classeur contient 5 feuilles
Application.SheetsInNewWorkbook = NbFeuilleDefaut
```

StatusBar (String)

Renvoie ou définit le texte de la barre d'état. Si vous lui affectez la valeur False, Excel reprend le contrôle du texte affiché.

```
Sub DemoStatus()

    Dim MaCell As Range, compteur As Long, SvgStatusText As String

    Application.StatusBar = False
    SvgStatusText = Application.StatusBar
    For Each MaCell In ThisWorkbook.Worksheets("Feuil1").Columns(1).Cells
        If IsNumeric(MaCell.Value) Then compteur = compteur + 1
        Application.StatusBar = "ligne " & MaCell.Row & " - Nombre valeur
num " & compteur
    Next
    Application.StatusBar = SvgStatusText
    Application.StatusBar = False

End Sub
```

WorksheetFunction (WorksheetFunction)

Cette propriété renvoie l'objet WorksheetFunction qui contient un certain nombre des fonctions intégrées d'Excel.

Nous n'allons pas les passer en détails ici puisqu'il y en a presque 200 dans Excel 2002. Les méthodes de l'objet WorksheetFunction porte le nom anglais des fonctions intégrées aussi n'est-il pas facile de faire toujours la correspondance. Le tableau suivant donne la liste des fonctions ainsi que leur équivalent anglais lorsqu'il est différent.

Acos	Acosh	Amorlin (Sln)
Arrondi (Round)	Arrondi.Inf (RoundDown)	Arrondi.Sup (RoundUp)
Asc	Asin	Asinh
Atan2	Atanh	BahtText
BdEcartType (DStDev)	BdEcartTypeP (DStDevP)	BdLire (DGet)
BdMax (DMax)	BdMin (DMin)	BdMoyenne (DAverage)
BdNb (DCount)	BdNbVal (DCountA)	BdProduit (DProduct)
BdSomme (DSum)	BdVar (DVar)	BdVarP (DVarP)
Beta.Inverse (BetaInv)	Centile (Percentile)	Centrée.Réduite (Standardize)
Cherche (Search)	ChercherB (SearchB)	Choisir (Choose)
Coefficient.Asymétrie (Skew)	Coefficient.Corrélation (Correl)	Coefficient.Détermination (RSq)
Combin	Cosh	Covariance (Covar)
Critère.Loi.Binomiale (CritBinom)	Croissance (Growth)	Ctxt (Fixed)
Db	Dbcs	DDb
Degrés (Degrees)	DéterMat (MDeterm)	DroiteReg (LinEst)
Ecart.Moyen (AveDev)	EcarType (StDev)	EcarTypeP (StDevP)
Epurage (Clean)	Equiv (Match)	Erreur.Type.Xy (StEyx)
EstErr (IsErr)	EstErreur (IsError)	EstLogique (IsLogical)
EstNA (IsNA)	EstNonTexte (IsNonText)	EstNum (IsNumber)
EstTexte (IsText)	Et (And)	Fact
Fisher	Fisher.Inverse (FisherInv)	Franc (Dollar)
Fréquence (Frequency)	Grande.Valeur (Large)	Impair (Odd)
Index	Intervalle.Confiance (Confidence)	Intper (IntRate)
Inverse.Loi.F (FInv)	InverseMat (MInverse)	Ispmt
Jours360 (Days360)	Joursem (Weekday)	Khideux.Inverse (ChiInv)
Kurtosis (Kurt)	Ln	LnGamma (GammaLn)
Log	Log10	LogReg (LogEst)
Loi.Beta (BetaDist)	Loi.Binomiale (BinomDist)	Loi.Binomiale.Nég (NegBinomDist)
Loi.Exponentielle (ExponDist)	Loi.F (FDist)	Loi.Gamma (GammaDist)
Loi.Gamma.Inverse (GammaInv)	Loi.Hypergéométrique (HypGeomDist)	Loi.Khideux (ChiDist)
Loi.LogNormale	Loi.LogNormale.Inverse	Loi.Normale

(LogNormDist)	(LogInv)	(NormDist)
Loi.Normale.Inverse (NormInv)	Loi.Normale.Standard (NormSDist)	Loi.Normale.Standard.Inverse (NormSInv)
Loi.Poisson (Poisson)	Loi.Student (TDist)	Loi.Student.Inverse (TInv)
Loi.Weibull (Weibull)	Max	Médiane (Median)
Min	Mode	Moyenne (Average)
Moyenne.Géométrique (GeoMean)	Moyenne.Harmonique (HarMean)	Moyenne.Réduite (TrimMean)
NB (Count)	Nb.Si (CountIf)	Nb.Vide (CountBlank)
NbVal (CountA)	NomPropre (Proper)	NPm (NPer)
Ordonnée.Origine (Intercept)	Ou (Or)	Pair (Even)
Pearson (Pearson)	Pente (Slope)	Permutation (Permut)
Petite.Valeur (Small)	Phonétique (Phonetic)	Pi
Plafond (Ceiling)	Plancher (Floor)	Prévision (Forecast)
Princper (Ppmt)	Probabilité (Prob)	Produit (Product)
ProduitMat (MMult)	Puissance (Power)	Quartile (Quartile)
Radians	Rang (Rank)	Rang.Pourcentage (PercentRank)
Recherche (Lookup)	RechercheH (HLookup)	RechercheV (VLookup)
Remplacer (Replace)	RemplacerB (ReplaceB)	Rept
Romain (Roman)	RTD	Sinh ()
Somme (Sum)	Somme.Carrés (SumSq)	Somme.Carrés.Ecarts (DevSq)
Somme.Si (SumIf)	Somme.X2MY2 (SumX2MY2)	Somme.X2PY2 (SumX2PY2)
Somme.XMY2 (SumXMY2)	SommeProd (SumProduct)	Sous.Total (Subtotal)
Substitue (Substitute)	SupprEspace (Trim)	Syd
Tanh	Taux (Rate)	Tendance (Trend)
Test.F (FTest)	Test.Khideux (ChiTest)	Test.Student (TTest)
Test.Z (ZTest)	Texte (Text)	Transpose
Tri (Irr)	TriM (MIrr)	Trouve (Find)
TrouverB (FindB)	USDollar	Va (Pv)
Van (Npv)	Var	Var.P (VarP)
Vc (Fv)	Vdb	Vpm (Pmt)

L'utilisation des méthodes de l'objet WorksheetFunction n'est pas toujours évidente car les arguments sont généralement faiblement typés et on n'a pas toujours le réflexe d'aller voir les paramètres attendus par la fonction intégrée correspondante. Prenons un exemple.

Vous avez besoin d'utiliser la fonction Excel EQUIV (Match) dans votre code. Dans l'explorateur d'objet, la méthode Match est définie comme :

Function Match(*Arg1, Arg2, [Arg3]*) **As Double**

Comme cela, il n'est pas évident de savoir à quoi correspondent les trois arguments. Par contre, si vous regardez l'aide sur Equiv, vous trouverez la définition

EQUIV(**valeur_cherchée**;**matrice_recherche**;type)

Ce qui est nettement plus compréhensible. Je peux donc utiliser un code tel que :

```
Dim Position As Long, MaFeuille As Worksheet

Set MaFeuille = ThisWorkbook.Worksheets("Feuil1")
With MaFeuille
    On Error Resume Next
    Position = Application.WorksheetFunction.Match(20, .Range(.Cells(1, 1),
.Cells(150, 1)), 0)
    If Err.Number = 1004 Then Err.Clear Else Err.Raise Err.Number,
Err.Source, Err.Description
    On Error GoTo 0
End With
```

Notez que je dois traiter l'erreur s'il n'y a pas de correspondance car si les fonctions intégrées renvoient des valeurs d'erreur, leurs homologues WorksheetFunction lèvent des erreurs récupérables.

Méthodes

Calculate

Permet de forcer le calcul. La syntaxe Application.Calculate est peu utilisée. On l'utilise principalement sous la forme [Worksheet].Calculate. Sachez toutefois que l'on peut restreindre le calcul à une plage à des fins de performance. Par exemple :

```
ThisWorkbook.Worksheets("Feuil1").Range("C1:C5").Calculate
```

Ne fait les calculs que sur la plage C1:C5.

ConvertFormula

De la forme :

Function ConvertFormula(*Formula* ***As*** *String*, *FromReferenceStyle* ***As*** *XlReferenceStyle*, [*ToReferenceStyle* ***As*** *XlReferenceStyle*], [*ToAbsolute* ***As*** *XlReferenceType*], [*RelativeTo As Range*]) **As String**

Permet de convertir une formule d'un système de référence à un autre et d'un mode à un autre.

Cette fonction peut s'avérer utile dans certain cas, cependant il y a souvent des façons plus simples d'arriver au même résultat.

L'exemple suivant convertit l'ensemble des formules de calcul de la feuille en référence absolue.

```
Sub ConvToAbsolute()

Dim MaFeuille As Worksheet, FormuleRel As String, FormuleAbs As String
Dim MaCell As Range

Set MaFeuille = ThisWorkbook.Worksheets("Feuil1")
For Each MaCell In MaFeuille.UsedRange.SpecialCells(xlCellTypeFormulas)
    FormuleRel = MaCell.Formula
    MaCell.Formula = Application.ConvertFormula(FormuleRel, xlA1, xlR1C1,
xlAbsolute)
Next

End Sub
```

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.

① 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)")
```

② 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.

La méthode Evaluate est extrêmement puissante puisqu'elle permet d'interpréter correctement à peu près toutes les expressions compréhensibles par Excel. Elle n'est toutefois pas évidente à bien manipuler, aussi doit on l'employer uniquement quand c'est la seule solution ou que son emploi est sans risque.

GetOpenFilename & GetSaveAsFilename

Ces deux méthodes permettent l'affichage d'une boîte de dialogue de sélection de fichiers renvoyant le ou les noms des fichiers sélectionnés. Ces méthodes n'ouvrent ou ne sauvent pas physiquement le fichier, elles ne renvoient que des noms. Elles renvoient False si l'utilisateur clique sur 'Annuler'.

Function GetOpenFilename([*FileFilter As String*], [*FilterIndex As Integer*], [*Title As String*], [*ButtonText*], [*MultiSelect As Boolean*]) **As String**

Function GetSaveAsFilename([*InitialFilename As String*], [*FileFilter As String*], [*FilterIndex As Integer*], [*Title As String*], [*ButtonText*]) **As String**

L'argument ButtonText n'est utilise que sous Macintosh.

En mode MultiSelect, la méthode GetOpenFilename renvoie un tableau de noms même si un seul nom est sélectionné, Faux si annuler est cliqué.

Le code suivant affiche la boîte de dialogue Enregistrer sous :

```
Dim NomFichier As String

    NomFichier = Application.GetSaveAsFilename(ThisWorkbook.FullName)
    If CBool(NomFichier) <> False Then
        ThisWorkbook.SaveAs NomFichier
    End If
```

Le code suivant propose une boîte de dialogue ouvrir permettant de sélectionner plusieurs fichiers.

```
Public Sub testboîte()

Dim NomFichier As Variant, Filtre As String, cmpt As Long

    Filtre = "Classeur (*.xls),*.xls, Fichiers texte (*.txt),*.txt, Tous
les fichiers(*.*),*.*"
    NomFichier = Application.GetOpenFilename(Filtre, 2, "Ouvrir", , True)
    If IsArray(NomFichier) Then
        For cmpt = LBound(NomFichier) To UBound(NomFichier)
            If StrComp(Right(NomFichier(cmpt), 3), "txt", vbTextCompare) =
0 Then
                Application.Workbooks.OpenText NomFichier(cmpt)
```

```
            ElseIf StrComp(Right(NomFichier(cmpt), 3), "xls",
vbTextCompare) = 0 Then
                Application.Workbooks.Open NomFichier(cmpt)
            End If
        Next cmpt
    End If

End Sub
```

InputBox

Il ne s'agit pas ici de la fonction VBA mais bien d'une méthode de l'objet Application. Il convient donc de faire attention à la notation, InputBox sans qualificateur désigne VBA.InputBox, et cette méthode doit être appelée comme Application.InputBox.

Function InputBox(*Prompt As String*, [*Title As String*], [*Default As Variant*], [*Left As Single*], [*Top As Single*], [*HelpFile As String*], [*HelpContextID As Long*], [*Type As Integer*]) **As Variant**

Mais pourquoi donc créer une méthode InputBox alors que le langage en fournit déjà une ?

La fonction InputBox de l'objet VBA présente quelques inconvénients.

- Elle renvoie une chaîne vide si l'utilisateur clique sur 'Annuler'
- Elle ne vérifie pas le type de la valeur saisie
- Elle ne permet pas de désigner une plage sur une feuille de calcul

C'est pour remédier à cela que l'objet Application propose une autre méthode InputBox. Pour tous ces premiers arguments, il s'agit de la même structure que la fonction InputBox, c'est-à-dire le texte d'invite, le titre, la valeur par défaut, les coordonnées de position et les identificateurs d'aide. En plus, on utilise l'argument Type précisant le(s) type(s) accepté(s). C'est un masque binaire définit comme :

Valeur	Type
0	Une formule.
1	Un nombre.
2	Texte (une chaîne).
4	Une valeur logique (True ou False).
8	Une référence de cellule, sous la forme d'un objet Range.
16	Une valeur d'erreur, telle que #N/A.
64	Un tableau de valeurs.

Cette méthode renverra Faux si vous cliquez sur 'Annuler'.

Si la saisie ne respecte pas l'argument type, Excel vous enverra un message d'avertissement et la boîte restera affichée. Attention quand même aux effets de bords.

```
    Dim IntVal As Integer
    IntVal = Application.InputBox(prompt:="chiffre", Type:=1)
```

Dans cet exemple, si l'utilisateur saisit des lettres, il aura l'affichage du message

Mais s'il clique sur 'Annuler', IntVal vaudra 0 car False sera convertit en zéro pas VBA.

Le type référence permet à l'utilisateur de sélectionner une zone sur la feuille.

Intersect & Union

Ces deux méthodes permettent de regrouper les objets Range passés en argument. Intersect renverra l'intersection des plages, c'est-à-dire l'ensemble des cellules appartenants à toutes les plages, ou Nothing si aucune cellule ne correspond.

```
Private Sub Worksheet_Change(ByVal Target As Range)

Dim PlageY As Range, PlageCible As Range

    Set PlageY = Me.Range("PlageY")
    Set PlageCible = Application.Intersect(PlageY, Target)
    If Not PlageCible Is Nothing Then
        For Each PlageY In PlageCible.Cells
            PlageY.AddComment "modifiée le " & Now
        Next
    End If

End Sub
```

Ce code ajoute un commentaire aux cellules modifiées de la plage nommée 'PlageY'.

Me : Qualificateur désignant l'objet qui le contient. Dans le module de code du classeur, **Me** désigne le classeur, dans le module de code d'une feuille, **Me** désigne la feuille. **Me** ne peut pas être employé dans un module standard.

Union renverra l'ensemble des plages. La fonction suivante renvoie l'ensemble des cellules contenant la valeur passée en argument.

```
Private Function FindValue(ByVal Valeur As Variant) As Range

    Dim Adresse As String, tmpFind As Range

    With ThisWorkbook.Worksheets("Feuil1").UsedRange
        Set FindValue = .Find(What:=Valeur, LookIn:=xlValues,
LookAt:=xlWhole)
        If FindValue Is Nothing Then Exit Function
        Adresse = FindValue.Address
        Set tmpFind = FindValue
        Do
            Set tmpFind = .FindNext(tmpFind)
            Set FindValue = Application.Union(FindValue, tmpFind)
        Loop While StrComp(tmpFind.Address, Adresse, vbTextCompare) = 0
    End With

End Fonction
```

Quit

Ferme la session Excel.

Workbooks & Workbook

La collection Workbooks contient une référence à l'ensemble des classeurs ouvert de la session en cours, l'objet Workbook désigne un classeur.

Manipuler la collection Workbooks

Propriété Item (Workbook)

De la forme :
Property Item(Index) **As Workbook**
Index étant un Variant représentant le numéro d'ordre ou le nom du classeur. Les numéros d'ordre étant gérés par Excel et pouvant varier d'une session à l'autre, on ne les utilise généralement pas. Le nom du classeur est un nom court de la forme "NomFichier.xls" si le classeur à déjà été enregistré, un nom sans extension si non. Le plus simple reste cependant de capturer la référence lors de l'ouverture ou de la création du classeur comme nous le verrons plus loin.

Propriété Count (Long)

Renvoie le nombre de classeurs ouverts dans la session Excel en cours. Les classeurs ne sont pas forcément visibles.

Méthode Add

Ajoute un nouveau classeur. De la forme :
Function Add([*Template*]) **As Workbook**
Où *Template* est un Variant suivant les règles suivantes :

- Si c'est une chaîne spécifiant un nom de fichier Excel existant, le nouveau classeur est créé sur le modèle du fichier spécifié.
- Si cet argument est une constante (*xlWBATChart*, *xlWBATExcel4IntlMacroSheet*, *xlWBATExcel4MacroSheet* ou *xlWBATWorksheet*), le nouveau classeur contient une seule feuille du type spécifié.
- Si *Template* est omis, Excel crée un nouveau classeur avec plusieurs feuilles de calcul dont le nombre est établi par la propriété *SheetsInNewWorkbook* de l'objet Application.

La méthode Add renvoie une référence à l'objet Workbook ajouté, c'est cette référence qu'il convient de capturer pour simplifier le code.

```
Sub test()

    Dim ClasseurGraphe As Workbook, ClasseurStandard As Workbook

    'ajoute un classeur contenant une feuille graphique
    Set ClasseurGraphe = Application.Workbooks.Add(xlWBATChart)
    'ajoute un classeur contenant des feuilles de calcul
    Set ClasseurStandard = Application.Workbooks.Add
    'les deux variables sont maintenant des qualificateurs manipulables
    MsgBox ClasseurGraphe.Name

End Sub
```

Méthode Close

Ferme tous les classeurs ouverts. Un message demandant s'il faut sauvegarder s'affichera le cas échéant.

Méthode Open

Ouvre un classeur dont le nom est passé comme premier argument. Le nom doit être un chemin long ou un chemin court s'il s'agit d'un classeur situé dans le répertoire courant. La méthode utilise de nombreux paramètres facultatifs comme la mise à jour des liens, le mot de passe ou le format que nous ne verrons pas ici. Là encore, une référence au classeur ouvert est renvoyé par la méthode.

```
Dim ClasseurSource As Workbook, ClasseurCible As Workbook

'ouvre un classeur avec un chemin absolu
Set ClasseurSource = Application.Workbooks.Open("d:\user\E221.xls")
'ouvre un classeur dans le répertoire en cours
Set ClasseurCible = Application.Workbooks.Open("d:\user\Recap.xls")
```

Méthode OpenText

Permet d'ouvrir un fichier texte sous forme de classeur Excel, les données étant éventuellement redistribuées en fonction des arguments passés. De la forme :

Sub OpenText(*Filename As String, [Origin As XlPlatform], [StartRow As Long], [DataType As XlTextParsingType], [TextQualifier As XlTextQualifier = xlTextQualifierDoubleQuote], [ConsecutiveDelimiter As Boolean], [Tab As Boolean], [Semicolon As Boolean], [Comma As Boolean], [Space As Boolean], [Other As Boolean], [OtherChar As Char], [FieldInfo As xlColumnDataType], [TextVisualLayout], [DecimalSeparator], [ThousandsSeparator], [TrailingMinusNumbers], [Local]*)

Pour bien appréhender le fonctionnement de cette procédure vous devez comprendre le fonctionnement de la redistribution dans Excel.

Par défaut, Excel considère comme saut de ligne les caractères retour chariot et ou saut de ligne. S'il n'en trouve pas, les données seront redistribuées sur une ligne. Vous ne pouvez pas indiquer d'autres caractères de saut de ligne directement à Excel, autrement dit, vous devrez traiter par code l'ouverture d'un fichier texte utilisant un séparateur différent.

Pour la répartition des données dans les colonnes, Excel comprend deux modes, la redistribution en largeur fixe et la redistribution délimitée. La redistribution en largeur fixe consiste à fixer une séparation par lot de n caractères sans tenir compte de la position d'un séparateur dans la ligne. Dans la redistribution délimitée, on définit un ou plusieurs séparateurs qui définissent le découpage en colonnes.

Évidemment, vous lèverez une erreur si le fichier contient plus de 65536 lignes ou si la division en colonnes implique plus de 256 colonnes.

C'est la définition de ce fonctionnement que gèrent les arguments de la méthode OpenText.

L'argument *Filename* est le seul argument obligatoire. Il désigne le fichier à ouvrir. Si aucun autre argument n'est passé, le fichier sera redistribué en lignes si un séparateur correspondant existe et en colonnes selon le caractère tabulation. Autrement dit, si le fichier texte ne contient ni caractère retour chariot ni caractère tabulation, Excel essayera de mettre la totalité du contenu dans la cellule A1. Si le contenu est trop grand, il sera automatiquement écrêté.

L'argument *StartRow* définit le numéro de ligne où devra commencer la redistribution dans le cas ou vous avez une entête que vous ne souhaitez pas redistribuer. S'il est omis, la redistribution commencera à la ligne 1.

L'argument *DataType* précise le mode de redistribution souhaitée, *xlDelimited* pour une redistribution délimitée, *xlFixedWidth* sinon.

L'argument *TextQualifier* définit le qualificateur de texte, généralement le caractère guillemet double (").

L'argument *ConsecutiveDelimiter* va définir le comportement de redistribution lorsqu'il existe plusieurs délimiteurs consécutifs. Si l'argument est vrai, les délimiteurs consécutifs seront considérés comme un seul, sinon, il sera laissé une cellule vide pour chaque délimiteur au delà du premier.

La liste des arguments suivants va définir les délimiteurs pour la redistribution en colonne. Les délimiteurs prédéfinis possible, sont dans l'ordre, Tabulation, point virgule, virgule, espace.

Vous pouvez définir un autre séparateur en mettant l'argument *Other* à vrai et en précisant le caractère dans l'argument *OtherChar*. Si celui-ci contient plus d'un caractère, seul le premier sera pris en compte.

L'argument *FieldInfo* est le plus complexe à manipuler. Il attend un tableau de tableaux de deux éléments. Les deux éléments se décomposent comme :

1) Le numéro d'ordre de la colonne en mode délimité ou la position du premier caractère de la colonne dans la ligne en mode largeur fixe
2) Le numéro de format de la colonne. Celle-ci est soit au format général (xlGeneralFormat) qui laisse Excel interpréter le type, soit au format texte (xlTextFormat), soit un format de date (xlMDYFormat, xlDMYFormat, xlYMDFormat, xlMYDFormat, xlDYMFormat, xlYDMFormat), soit un marqueur de non redistribution (xlSkipColumn).

En mode délimité, vous n'êtes pas obligé de préciser toutes les colonnes puisque par défaut les colonnes non précisées seront traitées sur la base du format général. Les éléments du tableau n'ont pas besoin d'être dans le même ordre que les colonnes.

En mode largeur fixe, vous ne pouvez pas omettre d'éléments.

Les autres arguments ne présentent pas de difficultés particulières.

Le code suivant va ouvrir un fichier texte en mode largeur fixe, ou chaque ligne sera redistribuée comme suit :

Une première colonne de huit caractères de format date
Une deuxième colonne de douze caractères au format texte
Une troisième colonne de cinq caractères au format texte
Une quatrième colonne de cinq caractères au format standard
Une cinquième colonne de quatre caractères au format standard
Une sixième colonne de trente six caractères au format texte
Tous les caractères au delà du soixante dixième seront ignorés

```
    Workbooks.OpenText Filename:="D:\User\Tutos\Excel\demo1.txt",
Origin:=xlMSDOS, StartRow:=1, DataType:=xlFixedWidth, _
    FieldInfo:=Array(Array(0, xlDMYFormat), _
            Array(8, xlTextFormat), _
            Array(20, xlTextFormat), _
            Array(25, xlGeneralFormat), _
            Array(30, xlGeneralFormat), _
            Array(34, xlTextFormat), _
            Array(70, xlSkipColumn)), TrailingMinusNumbers:=True
```

Le code suivant ouvrira un fichier texte en mode délimité ou les délimiteurs seront soit le point virgule, soit le point d'interrogation et où les délimiteurs successifs seront ignorés.

Les deux premières colonnes seront de type date et la quatorzième colonne sera ignorée.

La première ligne ne sera pas redistribuée.

```
    Workbooks.OpenText Filename:="D:\User\Tutos\Excel\demo1.txt",
Origin:=xlMSDOS, StartRow:=2, DataType:=xlDelimited,
TextQualifier:=xlDoubleQuote, _
        ConsecutiveDelimiter:=True, Tab:=False, Semicolon:=True,
Comma:=False, Space:=False, Other:=True, OtherChar:="?", _
        FieldInfo:=Array(Array(1, xlDMYFormat), _
                         Array(2, xlDMYFormat), _
                         Array(14, xlSkipColumn))
```

Propriétés de l'objet Workbook renvoyant une collection

BuiltinDocumentProperties

Renvoie la collection DocumentProperties qui contient des informations générales sur le classeur. Certaines de ses informations sont inhérentes au classeur, d'autres doivent avoir été saisie. S'utilise généralement sous la forme :

```
Workbook.BuiltinDocumentProperties(PropName).Value
```

Où *PropName* est le nom de la propriété recherchée. Il est aussi possible de passer l'index de l'élément recherché, mais vous prenez le risque d'avoir à modifier votre code selon la version d'Excel utilisée.

Les noms admissibles sont :

Title	Subject	Author
Keywords	Comments	Template
Last author	Revision number	Application name
Last print date	Creation date	Last save time
Total editing time	Number of pages	Number of words
Number of characters	Security	Category
Format	Manager	Company
Number of bytes	Number of lines	Number of paragraphs
Number of slides	Number of notes	Number of hidden Slides
Number of multimedia clips	Hyperlink base	Number of characters (with spaces)

Le code suivant renvoie à chaque enregistrement une boîte de messages avec l'auteur, le titre et la date de sauvegarde.

```
Private Sub Workbook_BeforeSave(ByVal SaveAsUI As Boolean, Cancel As
Boolean)

    Dim Nom As String, Titre As String, DateSave As Date
    Dim msg As String

    msg = "Classeur " &
ThisWorkbook.BuiltinDocumentProperties("Title").Value & vbCrLf
    msg = msg & "Auteur " &
ThisWorkbook.BuiltinDocumentProperties("Author").Value & vbCrLf
    msg = msg & "Sauvé le " & ThisWorkbook.BuiltinDocumentProperties("Last
save time").Value & vbCrLf
    MsgBox msg

End Sub
```

Il existe une collection **CustomDocumentProperties** qui permet de gérer des propriétés de votre choix.

Charts

Renvoie la collection des feuilles graphiques du classeur. Les graphiques incorporés aux feuilles de calcul ne sont pas membres de cette collection.

Names

Renvoie la collection des noms du classeur. Attardons nous un peu sur cette notion de nom dans Excel. Les noms dans Excel sont des variables locales attribuées généralement à un classeur (parfois à une feuille) et sauvegardées avec lui. Un nom Excel fait donc référence soit à une cellule ou plage de cellule (cas le plus courant) mais peut aussi faire référence à une formule ou à une valeur.

La collection Names se manipule un peu différemment des autres collections. Pour ajouter un élément à la collection, on peut utiliser la méthode Add. Celle-ci est de la forme :

Function Add(*[Name], [RefersTo], [Visible], [MacroType], [ShortcutKey], [Category], [NameLocal], [RefersToLocal], [CategoryLocal], [RefersToR1C1], [RefersToR1C1Local]*) **As Name**

Vous noterez déjà que, fait rare, tous les arguments sont facultatifs. Cela ne veut pas dire qu'aucun n'est obligatoire. Dans le concept des noms Excel, vous pouvez décider de travailler en version dite localisée ou en version standard. A minima, un nom doit donc avoir un nom défini dans l'argument *Name* ou dans l'argument *NameLocal*, et une référence défini dans un des arguments *RefersTo*.

Vous pouvez choisir de masquer le nom en mettant l'argument *Visible* à Faux. Je ne détaillerais pas ici les notions de type, de macro, ni de catégorie.

Attention, pour faire référence à une plage de cellule à ne pas omettre le signe égal en début de référence.

```
Dim Maplage As Range

'le nom réfère à une plage de cellule
ThisWorkbook.Names.Add NameLocal:="Plage", RefersToLocal:="=A1:B15"
'le nom réfère à une chaîne de caractère
ThisWorkbook.Names.Add NameLocal:="Chaîne", RefersToLocal:="A1:B15"
'ajoute un nom par action sur la propriété Name de l'objet Range
Set Maplage = ThisWorkbook.Worksheets("Feuil1").Range("A1:A10")
Maplage.Name = "Nom1"
```

Ce code vous montre aussi une façon indirecte d'ajouter un nom à la collection Names par le biais d'un objet Range.

Pour extraire un nom de la collection, on utilise la méthode Item en passant comme argument soit le nom en premier argument soit la valeur de la référence en troisième argument. Dans l'exemple suivant, la boîte de messages va afficher deux fois le même index.

```
ThisWorkbook.Names.Add "Exemple", 25
Dim MonNom As Name
Set MonNom = ThisWorkbook.Names("exemple")
MsgBox MonNom.Index
Set MonNom = ThisWorkbook.Names(, , "=25")
MsgBox MonNom.Index
```

Notez que pour l'appel par la valeur de référence, je suis obligé d'utiliser la valeur stricte des références de nom Excel, c'est-à-dire une chaîne composée du signe égal suivi de la référence.

Pour récupérer la valeur d'un nom, il suffit donc d'appeler une des propriétés RefersTo de l'objet Name sélectionné. Notez que si la référence est une plage de cellule, je peux renvoyer un objet Range avec la propriété RefersToRange ou même directement passer le nom comme argument de l'objet Range.

```
Dim Maplage As Range

'le nom réfère à une plage de cellule
ThisWorkbook.Names.Add NameLocal:="Plage", RefersToLocal:="=A1:B15"
Set Maplage = ThisWorkbook.Names("Plage").RefersToRange
Set Maplage = ThisWorkbook.Worksheets("Feuil1").Range("Plage")
```

Attention lorsque vous affectez des valeurs à des noms, la valeur renvoyée par les propriétés RefersTo seront de la forme chaîne commençant par égal. Vous pouvez utiliser la méthode Evaluate de l'objet application pour reconvertir la valeur dans le bon type.

```
ThisWorkbook.Names.Add "Exemple", 25
'Lève une erreur de type
MsgBox 2 * ThisWorkbook.Names("Exemple").RefersTo
'Affiche 50
MsgBox 2 * Application.Evaluate(ThisWorkbook.Names("Exemple").RefersTo)
```

Pour retirer un nom de la collection on utilise la méthode Delete sur un nom défini.

Sheets

Comme nous l'avons déjà vu, la collection Sheets est assez piégeuse, le seul moment où il est utile de l'utiliser est pour extraire le nombre de feuille du classeur ou pour connaître le type d'une feuille du classeur.

```
Sub TestFeuille()

Dim NbFeuille As Integer, compteur As Long, TypeFeuille As String

    NbFeuille = ThisWorkbook.Sheets.Count
    For compteur = 1 To NbFeuille
        Select Case ThisWorkbook.Sheets(compteur).Type
            Case XlSheetType.xlChart
                TypeFeuille = "Feuille graphique"

            Case XlSheetType.xlDialogSheet
                TypeFeuille = "Boîte de dialogue Excel 5"

            Case XlSheetType.xlExcel4IntlMacroSheet
                TypeFeuille = "Feuille Macro internationale Excel 4"

            Case XlSheetType.xlExcel4MacroSheet
                TypeFeuille = "Feuille macro excel 4"

            Case XlSheetType.xlWorksheet
                TypeFeuille = "Feuille de calcul"
        End Select
        MsgBox
    Next compteur

End Sub
```

Worksheets

Renvoie la collection des feuilles de calcul du classeur.

Quelques propriétés & méthodes de l'objet Workbook

Bien que l'objet Workbook expose de nombreuses propriétés et méthodes, elles sont assez spécifiques et plutôt utilisées dans des scénarii particuliers ; nous n'en verrons ici que quelques-unes.

Propriétés FullName, Name & Path (String)

Renvoie l'ensemble ou les éléments constitutifs du chemin du classeur.
FullName renvoie le nom complet du classeur
Name le nom du classeur avec son extension
Path le chemin (sans le dernier séparateur de fichier)

Propriété ReadOnly (Boolean)

Renvoie vrai si le classeur est ouvert en lecture seule.

Propriété Saved (Boolean)

Renvoi ou défini si le classeur a été modifié depuis la dernière sauvegarde. Vous pouvez mettre cette propriété à False même si le classeur a été modifié pour ne pas avoir l'affichage du message d'alerte Excel lors de la fermeture d'un classeur modifié.

Méthode Close

Ferme le classeur.

Sub Close(*[SaveChanges As Boolean], [Filename As String],…*)

Où *SaveChanges* définit la stratégie de sauvegarde telle que :

- Les modifications sont enregistrées si l'argument est vrai.
- Les modifications ne sont pas enregistrées si l'argument est faux.
- Une boîte de dialogues demande s'il faut enregistrer les changements le cas échéant s'affiche.

Et *Filename* définit un nom de fichier pour la sauvegarde, identique dans ce cas à une utilisation de SaveAs (Enregistrer sous).

```
    If Not ThisWorkbook.Saved Then
        ThisWorkbook.Close True
    End If
```

Méthode Protect

Protège le classeur.

Sub Protect(*[Password As String], [Structure As Boolean], [Windows As Boolean]*)

L'argument *Password* est le mot de passe de protection. S'il est omis, le classeur est protégé sans mot de passe et il est alors possible de le déprotéger à partir du menu Excel. Si l'argument *Structure* est vrai, l'ordre des feuilles est fixe et il n'est pas possible d'en ajouter ou d'en supprimer, si l'argument *Windows* est vrai, les fenêtres affichant le classeur sont bloquées.

Notez que vous pouvez appeler Protect sans argument et définir les propriétés équivalentes Password, ProtectStructure et ProtectWindows.

Ne confondez pas le mot de passe de protection du classeur avec celui restreignant l'accès en écriture.

Méthodes Save, SaveAs & SaveCopyAs

Ces trois méthodes gèrent la sauvegarde du classeur.

La méthode Save n'attend pas d'argument. Si vous l'utilisez sur un classeur que vous venez de créer et qui n'a jamais été sauvegardé, Excel tentera de l'enregistrer sous le nom donné par défaut à la création suivi de l'extension xls dans le répertoire courant.

La méthode **SaveCopyAs** attend un nom de fichier comme argument. Le classeur ouvert ne verra pas sa propriété **Saved** ramené à vraie après l'appel de la méthode **SaveCopyAs** puisqu'il n'aura pas été enregistré.

```
Sub TestSave()

Dim MonClasseur As Workbook

    Set MonClasseur = Application.Workbooks.Add
    With MonClasseur.Worksheets(1).Cells(1, 1)
        .Value = 1
        .Resize(100).DataSeries Rowcol:=xlColumns, Type:=xlLinear, Step:=1,
Stop:=100, Trend:=False
    End With
    MonClasseur.SaveCopyAs "temp.xls"
    Debug.Print MonClasseur.Saved
    'Faux
End Sub
```

La méthode **SaveAs** est de la forme :

Sub SaveAs([Filename], [FileFormat], [Password], [WriteResPassword], [ReadOnlyRecommended], [CreateBackup], [AccessMode As XlSaveAsAccessMode = xlNoChange], [ConflictResolution], [AddToMru], [TextCodepage], [TextVisualLayout], [Local])

```
Sub TestSave()

Dim MonClasseur As Workbook

    Set MonClasseur = Application.Workbooks.Add
    With MonClasseur.Worksheets(1).Cells(1, 1)
        .Value = 1
        .Resize(100).DataSeries Rowcol:=xlColumns, Type:=xlLinear, Step:=1,
Stop:=100, Trend:=False
    End With
    MonClasseur.SaveAs Filename:="D:\User\MonClasseur.xls",
FileFormat:=xlNormal, Password:="", WriteResPassword:="password",
ReadOnlyRecommended:=True, CreateBackup:=False

End Sub
```

Méthode Unprotect

De la forme Classeur.Unprotect("Mot de passe"), retire la protection du classeur.

Worksheets & Worksheet

La collection Worksheets représente l'ensemble des feuilles de calcul du classeur. Elle se manipule comme l'ensemble des collections. Notez toutefois que la propriété Item renverra un objet et non un objet Worksheet. Vous devrez forcer la conversion pour éviter d'avoir une rupture IntelliSense.

Méthodes de la collection Worksheets

Add

Permet d'ajouter une feuille à la collection en précisant la position d'ajout.

Function Add(*[Before As Sheet], [After As Sheet], [Count As Integer], [Type As XlSheetType]*) **As Object**

Je vous ai marqué **Sheet** comme type d'argument pour *Before* et *After* bien que ce type n'existe pas. En effet, ces arguments attendent une feuille précise du classeur pour savoir ou insérer la feuille créée. Le comportement peut être différent selon la nature de la précision. Imaginons un classeur contenant une feuille de calcul, une feuille graphique puis une autre feuille de calcul.

```
Sub AddWorksheet()

    Dim NouvelleFeuille As Worksheet
    Set NouvelleFeuille =
ThisWorkbook.Worksheets.Add(after:=ThisWorkbook.Worksheets(2))
    'la feuille sera insérée après la deuxième feuille de calcul (4éme
position)
    Set NouvelleFeuille =
ThisWorkbook.Worksheets.Add(after:=ThisWorkbook.Sheets(2))
    'la feuille sera insérée après la deuxième feuille (donc graphique)
(3éme position)

End Sub
```

Vous ne devez préciser qu'un des deux arguments, soit *After*, soit *Before*, si vous n'en précisez aucun la feuille sera ajoutée en première position.

L'argument Count permet d'ajouter plusieurs feuilles ensemble.

L'argument Type est une aberration. Il permet de préciser le type de feuille que l'on souhaite ajouter. En toute logique, puisque nous travaillons sur une collection de feuilles de calcul, nous ne devrions pouvoir ajouter que des feuilles de calcul. Mais si nous écrivons

```
ThisWorkbook.Worksheets.Add After:=ThisWorkbook.Worksheets(2),
Type:=xlChart
```

Le compilateur ne verra pas d'erreurs avant que vous n'exécutiez la ligne. Là, une erreur sera levée, puisque ce n'est pas cohérent, mais cela aurait été plus simple de ne pas mettre l'argument Type.

Copy

Permet de copier une feuille dans le classeur défini à la position défini. De la forme :
Sub Copy(*[Before], [After]*)
Les arguments Before et After fonctionnent comme pour la méthode Add. Ils ne désignent cependant pas obligatoirement une feuille du classeur dont est issue la feuille à copier. Ceci implique qu'avec cette méthode on peut ou bien dupliquer la feuille dans un même classeur, ou copier la feuille dans un autre classeur.

```
Sub CopyWorksheet()

    'duplique la feuille en dernière position du classeur d'origine
    ThisWorkbook.Worksheets("Feuil1").Copy
After:=ThisWorkbook.Sheets(ThisWorkbook.Sheets.Count)
    Dim MonClasseur As Workbook
    Set MonClasseur = Application.Workbooks.Add
    'copie la feuille en première position dans un nouveau classeur
    ThisWorkbook.Worksheets("Feuil1").Copy
Before:=MonClasseur.Worksheets(1)

End Sub
```

Attention, si la feuille utilise des noms ou des références externes, la copier dans un nouveau classeur peut déclencher un certain nombre d'erreurs.

Notez que la feuille copiée ne renvoie pas de références sur la nouvelle feuille. La feuille copiée deviendra la feuille active ce qui vous donne un moyen éventuel de la retrouver, sinon vous pouvez le faire en jouant sur la position d'insertion.

```
Sub CopyWorksheet()

    Dim MonClasseur As Workbook, FeuilleCollee As Worksheet
    Set MonClasseur = Application.Workbooks.Add
    Set FeuilleCollee = MonClasseur.Worksheets(1)
    ThisWorkbook.Worksheets("Feuil1").Copy Before:=FeuilleCollee
    Set FeuilleCollee = FeuilleCollee.Previous

End Sub
```

Delete

Supprime l'ensemble des feuilles de calcul. Une exception sera levée si le classeur ne contient que des feuilles de calcul.

FillAcrossSheets

Permet de recopier une plage de cellules dans l'ensemble des feuilles sélectionnées. De la forme :
Sub FillAcrossSheets(*Range As Range, [Type As XlFillWith = xlFillWithAll]*)
Pour désigner une sélection de feuille sans avoir à les sélectionner réellement, on passe un tableau de noms de feuilles à la propriété Worksheets.

```
Dim ListFeuille As Variant

ListFeuille = Array("Feuil1", "Feuil2", "Feuil4")
ThisWorkbook.Worksheets(ListFeuille).FillAcrossSheets _
Worksheets("Feuil1").Range("A1:A20"), xlFillWithContents
```

Move

Déplace la feuille dans le classeur

Sub Move(*[Before], [After]*)

Où *Before* et *After* désigne une feuille de référence.

Par exemple, le code suivant passe la première feuille de calcul en dernière position (des feuilles de calcul)

```
Sub MoveSheet2End()

    ThisWorkbook.Worksheets(1).Move
after:=ThisWorkbook.Worksheets(ThisWorkbook.Worksheets.Count)

End Sub
```

PrintOut

Ordonne l'impression de la feuille, telle que

Sub PrintOut(*[From], [To], [Copies], [Preview], [ActivePrinter], [PrintToFile], [Collate], [PrToFileName]*)

Je ne rentrerais ici dans le détail des arguments car généralement l'appel de la méthode sans argument suffit.

Propriétés de l'objet Worksheet renvoyant une collection

Cells

Renvoie la collection des cellules de la feuille sous forme d'un objet Range.

Columns & Rows

Renvoie la collection des colonnes ou des lignes de la feuille sous la forme d'un objet **Range**. Je le mets en gras car il n'existe pas dans Excel d'objet Column ou Row. On peut récupérer une colonne ou une ligne soit en passant par l'identificateur de la colonne ou de la ligne, soit en utilisant l'index, soit par des méthodes dérivées. Nous traiterons de tout cela plus loin dans le présent document.

Comments

Bien que les commentaires soient attribués à une cellule, ils sont accessibles par cette propriété qui renvoie tous les commentaires des cellules de la feuille.

On utilise plus souvent la propriété Comment de l'objet Range pour atteindre un commentaire spécifique ou par la méthode SpecialCells pour les récupérer tous.

Hyperlinks

Renvoie la collection des liens hypertexte de la feuille. Un lien hypertexte est généralement affecté à une cellule ou à un objet, il désigne un emplacement dans le classeur ou dans un document externe, une adresse mail, etc.…

```
Sub TestHyperTexte()
    Dim MaFeuille As Worksheet, MonLien As Hyperlink

    Set MaFeuille = ThisWorkbook.Worksheets("Feuil2")
    Set MonLien = MaFeuille.Hyperlinks.Add(Anchor:=MaFeuille.Cells(2, 2),
Address:= _
        "D:\JMARC\Classeur1.xls", SubAddress:="Feuil1!L1C1:L10C5",
TextToDisplay:= _
        "Liste des données externes")
    MonLien.Follow
End Sub
```

Names

Renvoie la collection des noms définis dans **la feuille de calcul**. Normalement les noms Excel sont définis au niveau du classeur. Il est cependant possible de définir un nom au niveau de la feuille, généralement dans les deux cas suivants :

- On ne souhaite pas que le nom soit visible à l'extérieur de la feuille
- On veut redéfinir un nom existant dans le classeur pour la feuille.

Shapes

Renvoie la collection des objets non OLE contenus dans la feuille, à savoir formes géométriques, contrôles de type formulaire, images, etc…

Nous ne traiterons pas de ces objets dans ce cours, sauf pour les contrôles incorporés.

Autres propriétés de l'objet Worksheet

FilterMode (Boolean)

Renvoie vrai si la feuille est en mode filtré. Nous verrons un peu plus loin l'utilisation des filtres.

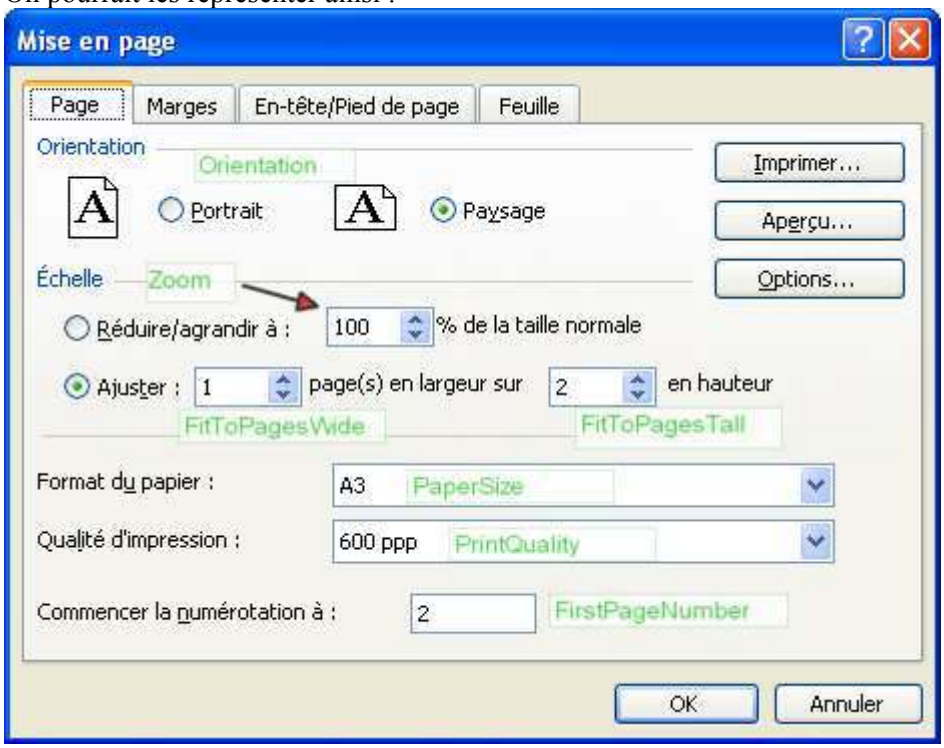
Next & Previous (Worksheet)

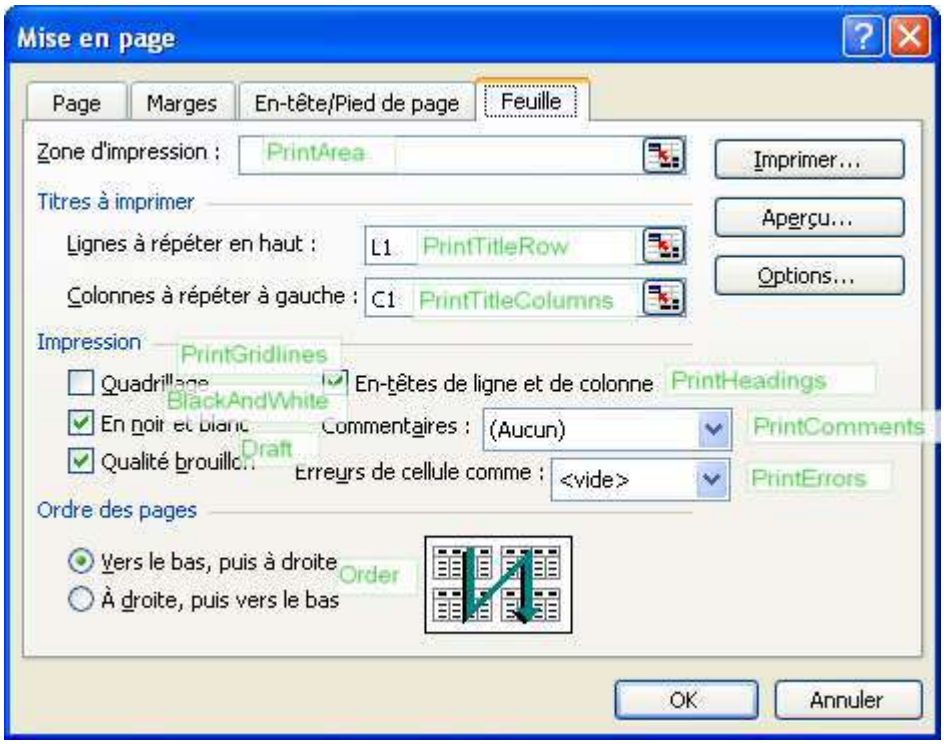
Renvoie la feuille suivante ou précédente dans la collection des feuilles de calcul ou Nothing si on appelle Previous sur la première feuille de la collection ou Next sur la dernière.

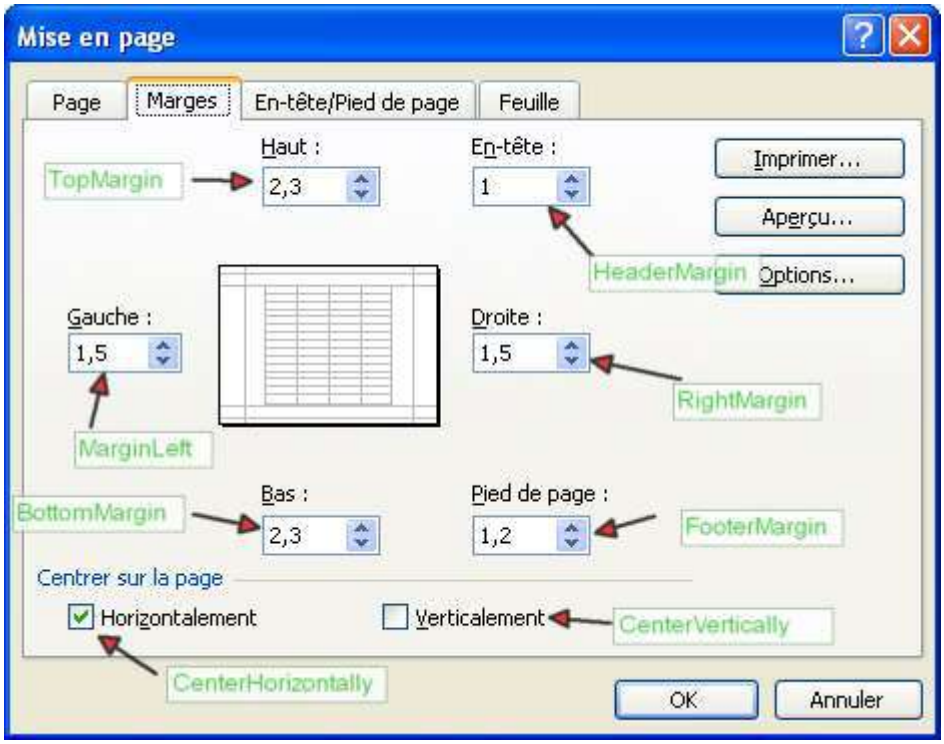
PageSetup (PageSetup)

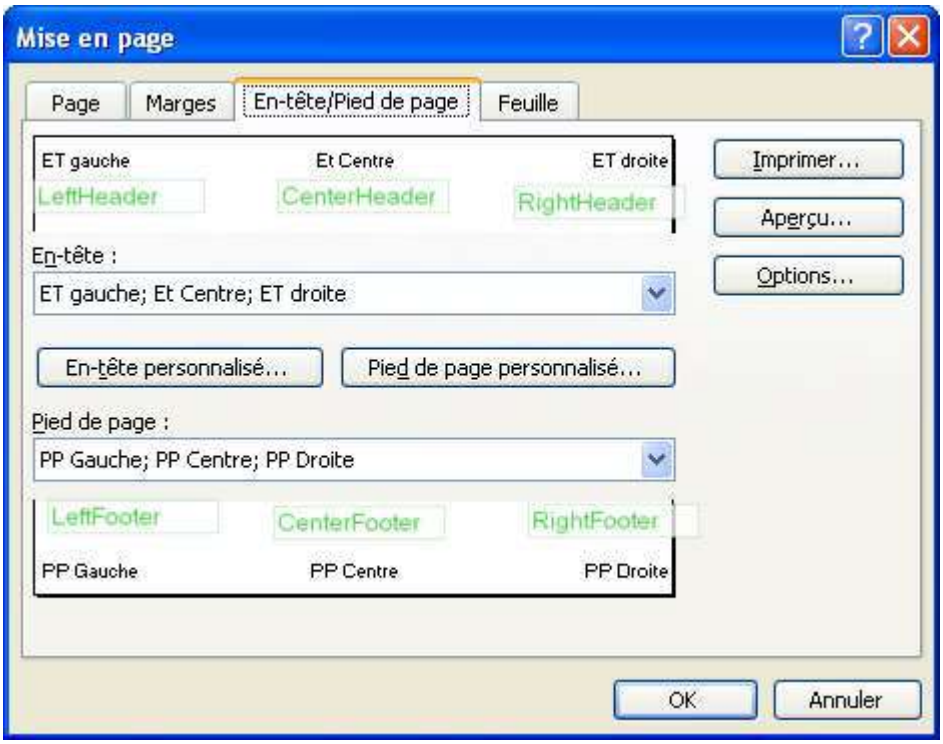
L'objet PageSetup renvoyé représente la mise en page de la feuille généralement à fin d'impression. Les propriétés de l'objet PageSetup correspondent aux onglets de la boîte de dialogue **mise en page** du menu **Fichier**.

On pourrait les représenter ainsi :




Mise en page
Page
Marges
En-tête/Pied de page
Feuille
Zone d'impression :
PrintArea
Imprimer...
Aperçu...
Options...
Titres à imprimer
Lignes à répéter en haut :
L1
PrintTitleRow
Colonnes à répéter à gauche :
C1
PrintTitleColumns
Impression
PrintGridlines
Quadrillage
En-têtes de ligne et de colonne
PrintHeadings
BlackAndWhite
En noir et blanc
Commentaires :
(Aucun)
PrintComments
Draft
Qualité brouillon
Erreurs de cellule comme :
<vide>
PrintErrors
Ordre des pages
Vers le bas, puis à droite
Order
À droite, puis vers le bas
OK
Annuler


Mise en page
Page
Marges
En-tête/Pied de page
Feuille
Haut :
TopMargin
2,3
En-tête :
1
Imprimer...
Aperçu...
HeaderMargin
Options...
Gauche :
1,5
Droite :
1,5
RightMargin
MarginLeft
BottomMargin
Bas :
2,3
Pied de page :
1,2
FooterMargin
Centrer sur la page
Horizontalement
Verticalement
CenterVertically
CenterHorizontally
OK
Annuler



Ce qui en code VBA s'écrirait

```
With ActiveSheet.PageSetup
        .PrintTitleRows = "$1:$1"
        .PrintTitleColumns = "$A:$A"
        .LeftHeader = "ET gauche"
        .CenterHeader = "ET Centre"
        .RightHeader = "ET droite"
        .LeftFooter = "PP Gauche"
        .CenterFooter = "PP Centre"
        .RightFooter = "PP Droite"
        .LeftMargin = Application.InchesToPoints(0.590551181102362)
        .RightMargin = Application.InchesToPoints(0.590551181102362)
        .TopMargin = Application.InchesToPoints(0.905511811023622)
        .BottomMargin = Application.InchesToPoints(0.905511811023622)
        .HeaderMargin = Application.InchesToPoints(0.393700787401575)
        .FooterMargin = Application.InchesToPoints(0.47244094488189)
        .PrintHeadings = True
        .PrintGridlines = False
        .PrintComments = xlPrintNoComments
        .PrintQuality = 600
        .CenterHorizontally = True
        .CenterVertically = False
        .Orientation = xlLandscape
        .Draft = True
        .PaperSize = xlPaperA3
        .FirstPageNumber = 2
        .Order = xlDownThenOver
        .BlackAndWhite = True
        .Zoom = False
        .FitToPagesWide = 1
        .FitToPagesTall = 2
        .PrintErrors = xlPrintErrorsBlank
    End With
```

Range (Range)

Renvoie ou définit un groupe de cellules de la feuille.

UsedRange (Range)

Renvoie ou définit la plage rectangulaire des cellules utilisées. Attention il ne s'agit pas de la plage des cellules contenant quelque chose, mais une plage allant de la première cellule à la dernière cellule ayant un contenu ou une propriété différant du standard.

```
Private Sub testusedrange()

    Dim MaFeuille As Worksheet

    Set MaFeuille = ThisWorkbook.Worksheets.Add
    MaFeuille.Cells(100, 10).Style = "Pourcentage"
    Debug.Print MaFeuille.UsedRange.AddressLocal(True, True, xlR1C1)
    'L100C10
    MaFeuille.Cells(2, 3).Value = "test"
    Debug.Print MaFeuille.UsedRange.AddressLocal(True, True, xlR1C1)
    'L2C3:L100C10

End Sub
```

On utilise fréquemment UsedRange lorsqu'on doit travailler sur les cellules de la feuille pour restreindre la portée aux cellules réellement utilisées.

Visible (XlSheetVisibility)

Définit l'affichage ou le masquage de la feuille. Peut prendre une des valeurs suivantes :

xlSheetHidden	**0**	Masque la feuille
xlSheetVisible	**-1**	Affiche la feuille
xlSheetVeryHidden	**2**	Masque la feuille sans qu'il soit possible pour l'utilisateur de la faire afficher.

Méthodes de l'objet Worksheet

Calculate

Force le calcul de la feuille. Inutile lorsque le mode de calcul est sur automatique. N'oubliez pas que bloquer le calcul automatique peut accélérer l'exécution du code.

ChartObjects

Renvoie soit un graphique incorporé, soit la collection des graphiques incorporés de la feuille. Nous verrons cela en détail dans l'étude des graphiques.

Copy

Identique à la méthode de la collection Worksheets

Delete

Supprime la feuille. Sauf si vous avez désactivé la propriété DisplayAlerts de l'objet Application, vous aurez l'affichage d'un message de confirmation de suppression.

Move

Identique à la méthode de la collection Worksheets

OLEObjects

Renvoie la collection des objets OLE contenus dans la feuille. Sans entrer trop dans le détail, les objets OLE sont des éléments intégrés fournis par des composants externes (ou d'autres applications) pouvant être édités dans l'application qui les a créée. On trouve, entre autre, dans cette collection, les contrôles de boîte de dialogues intégrés dans la feuille, ainsi que les objets accessible depuis le menu Objet dans le menu Insertion d'Excel.

```
Sub InsertOLE()

    Dim Mafeuille As Worksheet

    Set Mafeuille = ThisWorkbook.Worksheets("Feuil3")
    'ajoute un fichier word existant et l'intègre comme objet lié
    Mafeuille.OLEObjects.Add Filename:="D:\tutoriel\tuto.doc", Link:=True,
DisplayAsIcon:=False
    'ajoute un controle spin comme objet embarqué
    Mafeuille.OLEObjects.Add ClassType:="Forms.SpinButton.1", Link:=False,
DisplayAsIcon:=False, Left:=173.25, Top:=197.25, Width:=12.75, Height:=25.5

End Sub
```

Paste & PasteSpecial

Ces deux méthodes servent à copier des éléments sur une feuille de calcul et non une feuille de calcul elle-même. De la forme :

Sub Paste(*[Destination As Range], [Link As Boolean]*)

Où destination est la plage où doit être collé le contenu du presse-papiers. Si l'argument est omis et qu'une plage est nécessaire, c'est la sélection qui sera utilisée. L'argument Link colle l'élément comme un lien Hypertexte. Les deux arguments sont exclusifs, vous devez préciser soit l'un, soit l'autre, soit aucun.

Sub PasteSpecial(*[Format], [Link], [DisplayAsIcon], [IconFileName], [IconIndex], [IconLabel], [NoHTMLFormatting]*)

Ne confondez pas cette méthode avec la méthode du même nom de l'objet Range. La méthode PasteSpecial de l'objet Worksheet colle des éléments externes à Excel, généralement des objets OLE.

De manière générale, on évite d'utiliser ces méthodes pour coller des cellules Excel, on utilise plutôt la méthode Copy de l'objet Range ou des affectations directes.

PrintOut

Identique à la méthode de la collection Worksheets

Protect & Unprotect

Protège ou retire la protection de tout ou partie de la feuille.

Sub Protect(*[Password], [DrawingObjects], [Contents], [Scenarios], [UserInterfaceOnly], [AllowFormattingCells], [AllowFormattingColumns], [AllowFormattingRows], [AllowInsertingColumns], [AllowInsertingRows], [AllowInsertingHyperlinks], [AllowDeletingColumns], [AllowDeletingRows], [AllowSorting], [AllowFiltering], [AllowUsingPivotTables]*)

Le premier argument est le mot de passe de protection. S'il est omis, la feuille sera protégée mais la protection pourra être enlevée directement dans le menu Outils – Protection d'Excel.

Les autres arguments sont des booléens indiquant ce qui doit être protégé. La méthode **Unprotect** n'attend que le mot de passe comme argument, tous les éléments sont déprotégés quand on invoque **Unprotect**.

```
Dim Mafeuille As Worksheet

    Set Mafeuille = ThisWorkbook.Worksheets("Feuil1")
    Mafeuille.Protect "Password", DrawingObjects:=False, Contents:=True,
Scenarios:= _
        True, AllowInsertingColumns:=True, AllowInsertingRows:=True, _
        AllowInsertingHyperlinks:=True, AllowDeletingColumns:=True, _
        AllowDeletingRows:=True
```

Range & Cells

C'est dans la manipulation de ces objets que se trouve le corps de la programmation Excel. Lorsque vous saurez bien manipuler l'objet Range, la programmation VBA Excel ne présentera plus tellement de difficultés.

Pour ne pas allonger inutilement les codes exe rirais pas toujours les affectations des objets parent dans les exemples. Prenons comm able contenant le mot 'Feuille' sera un objet de type Worksheet et que toute variable contenant le mot 'Classeur' sera un objet de type Workbook.

Concepts

Le plus dur consiste à différencier Range et Cells. L'objet Range représente une ou plusieurs cellules de la même feuille de calcul. Lorsque l'objet Range représente plusieurs cellules, elles ne sont pas forcément contiguës, mais elles sont considérées comme une unité logique. C'est-à-dire que tout action sur l'objet Range agit sur toutes les cellules qu'il contient sauf si un sous ensemble particulier est spécifié.

Il n'existe pas d'objet Cells. Cells est une propriété qui renvoie une collection de cellules d'un objet contenant des cellules, c'est-à-dire d'un objet Worksheet ou d'un objet Range. La propriété Cells renvoie un objet Range puisque par définition, l'objet Range est une collection de cellules.

Comme vous le voyez, il n'existe pas de différence entre les deux, mais les deux sont représentés pour des raisons de lisibilité.

Le choix d'utiliser une notation Cells ou une notation Range est généralement dicté par la logique du code. Par exemple, pour faire référence à l'ensemble des cellules d'une feuille, j'aurais tendance à écrire :

```
Feuille.Cells
```

Mais pour faire référence à l'ensemble des cellules utilisées, je noterais :

```
Feuille.UsedRange
```

Dans les deux cas, c'est bien un objet Range qui est renvoyé.

Cela peut paraître incohérent, mais un objet Range expose toujours une collection Cells qui renvoie un objet Range identique.

De la même façon, les notations de la méthode Item de l'objet Range sont toutes équivalentes. Il n'y a pas de différences entre les références :

```
Feuille.Cells(2,3)
Feuille.Range("C2")
```

Même si dans certains cas une notation sera préférable pour des raisons d'écriture de code. Ainsi si nous souhaitons parcourir une cellule sur deux de la plage "A1:AZ1", il sera plus simple d'écrire :

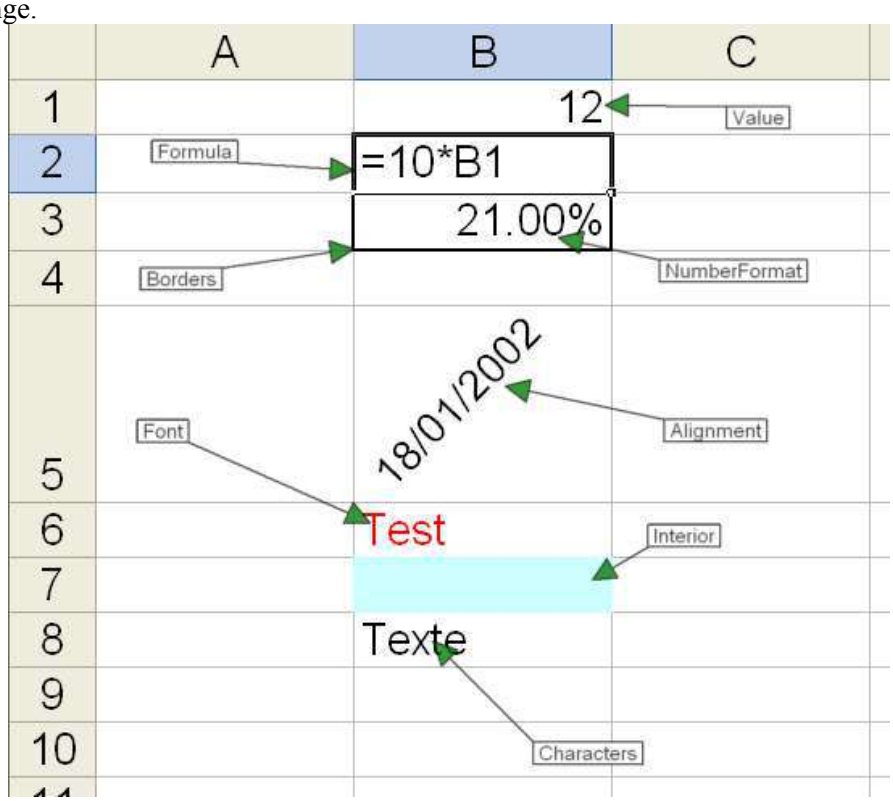
```
Sub Parcourir()

Dim compteur As Long, MaFeuille As Worksheet

Set MaFeuille = ThisWorkbook.Worksheets("Feuil1")
For compteur = 1 To 52 Step 2
    MaFeuille.Cells(1, compteur).Interior.Color = vbBlue
Next compteur

End Sub
```

Que

```
Sub Parcourir()

Dim compteur As Long, MaFeuille As Worksheet

Set MaFeuille = ThisWorkbook.Worksheets("Feuil1")
For compteur = 1 To 52 Step 2

MaFeuille.Range(Mid(MaFeuille.Columns(compteur).Address(columnAbsolute:=Fal
se), Len(MaFeuille.Columns(compteur).Address(columnAbsolute:=False)) \ 2 +
2) & "1").Interior.Color = vbBlue
Next compteur

End Sub
```

Le choix des notations vient avec l'expérience. Généralement, il est induit aussi par votre façon de manipuler les plages. Dans le présent cours, je vais utiliser ma façon de noter. Elle n'est pas universelle et en aucune façon la seule façon de procéder. Nous verrons plus loin dans l'étude de cas pratiques qu'il y a souvent plusieurs techniques envisageables pour un même but.

S'il est entendu que l'objet Cell n'existe pas en tant que tel, regardons quand même comment est architecturée une cellule au sens VBA du terme, c'est-à-dire sous la forme de propriétés de l'objet Range.



Et je n'ai représenté ici que quelques propriétés. Les cellules sont des objets relativement complexes, bien qu'on n'ait plus de difficultés à les manipuler lorsqu'on connaît le modèle.

Le deuxième aspect important de l'objet Range est sa façon d'agir sur plusieurs cellules comme s'il n'y en avait qu'une. Toutes les propriétés de l'objet Range ne réagissent pas de la même façon lorsque l'objet contient plusieurs cellules. Les propriétés de contenu, principalement Formula et Value, parfois appelées propriétés tableaux sont gérées sous forme d'un tableau de Variant. On peut alors les affecter directement à une plage de taille identique sans se soucier du contenu de chaque cellule, l'affectation se faisant par identification de position. Ainsi le code :

```
Range("C1:C5").Value = Range("A9:A13").Value
```

Recopiera bien les valeurs de la plage source dans la plage cible, où la valeur A9 sera recopiée dans C1, celle de A10 dans C2 et ainsi de suite.

Imaginons maintenant que nous ayons un code tel que :

```
Range("C1:C3").Interior.Color = Range("A9:A11").Interior.Color
```

Comment va-t-il être interprété ?

Et bien ça dépend. Comme la propriété Interior n'est pas une propriété tableau, Excel va la gérer comme une propriété regroupée. Les propriétés regroupées fonctionnent de la manière suivante :

- Si toutes les cellules de la plage ont la même valeur pour cette propriété, l'objet Range renverra cette valeur
- Si la propriété d'une cellule au moins diffère de celles des autres, l'objet Range renverra salon les cas, Null, Nothing, ou une valeur arbitraire.

Dans le cas de la propriété Color de l'objet Interior, la valeur renvoyée pour une plage hétérogène est la valeur 16777215 (&FFFFFF) (blanc). Autrement dit, si toutes les cellules de la plage A9:A11 ont la même couleur de fond, alors la plage C1:C3 aura aussi la même couleur, sinon, la plage C1:C3 aura un fond blanc. Il est toujours assez complexe de savoir ce que renvoient les propriétés regroupées, on évite donc généralement de procéder comme dans mon exemple précédent. En affectation il ne peut y avoir de problèmes, du fait que la modification de la valeur de la propriété de la plage induit la modification de la propriété pour chaque cellule pour une valeur identique. En lecture, il est indispensable de savoir qu'elle valeur sera renvoyée pour les plages hétérogènes. Dans l'étude des propriétés, je vous noterais cette valeur sous la forme :

Plage hétérogène ⇨ Valeur

Nous allons voir maintenant les propriétés et méthodes de l'objet Range qui sont donc autant celles d'une cellule que d'une collection de cellules. A chaque fois que j'emploierais le terme la plage de cellules, cela s'appliquera aussi à une cellule unique, sauf spécification particulière.

Valeurs & Formules

Nous allons commencer par traiter des propriétés assez particulières de l'objet Range qui gèrent le contenu de la cellule.

Pour Excel, le contenu des cellules n'est pas typé à priori. Excel interprète le contenu des cellules et peut selon les cas, mettre la donnée sous une autre forme ou appliquer un formatage ce qui fait que l'apparence du contenu peut différer du contenu réel.

Notez que dans Excel, le contenu de la cellule tel qu'il est entré par l'utilisateur n'existe pas forcément dans une des propriétés de la cellule.

Comme Excel est un tableur, il gère aussi des formules. Dans ce cas, le contenu réel de la cellule est la formule. Cependant, les mêmes règles s'appliquent. C'est-à-dire que le résultat de la formule est calculé, et que ce contenu peut être formaté. Ce mécanisme qui paraît assez évident lorsqu'on l'utilise directement est assez complexe au niveau des propriétés de l'objet Range.

Celui possède donc plusieurs propriétés qui exposent les différentes représentations du contenu.

Celles-ci se regroupent globalement en deux familles, celles qui exposent les formules :

- Formula
- FormulaR1C1
- FormulaLocal
- Formula R1C1Local

Celles qui exposent les valeurs.

- Value
- Value2
- Text

Cette division n'a de sens que lorsque la cellule contient une formule, lorsqu'elle contient une valeur, les propriétés de formule renvoient la valeur. Lorsqu'une cellule est vide, elle renvoie *Empty*.

Prenons un exemple simple, la cellule A1 contient la valeur 14, la cellule A2 contient la formule =2*A1/10, la cellule A3 contient la formule =Log(A1;A2)

Si on lit à l'aide d'un code les propriétés précédentes, on obtient :

```
Debug.Print Range("A1").Formula '14
Debug.Print Range("A1").FormulaLocal '14
Debug.Print Range("A1").FormulaR1C1 '14
Debug.Print Range("A1").FormulaR1C1Local '14
Debug.Print Range("A1").Value '14
Debug.Print Range("A1").Value2 '14
Debug.Print Range("A1").Text '14
Debug.Print Range("A2").Formula '=2*A1/10
Debug.Print Range("A2").FormulaLocal '=2*A1/10
Debug.Print Range("A2").FormulaR1C1 '=2*R[-1]C/10
Debug.Print Range("A2").FormulaR1C1Local '=2*L(-1)C/10
Debug.Print Range("A2").Value ' 2,8
Debug.Print Range("A2").Value2 ' 2,8
Debug.Print Range("A2").Text '2,8
Debug.Print Range("A3").Formula '=LOG(A1,A2)
Debug.Print Range("A3").FormulaLocal '=LOG(A1;A2)
Debug.Print Range("A3").FormulaR1C1 '=LOG(R[-2]C,R[-1]C)
Debug.Print Range("A3").FormulaR1C1Local '=LOG(L(-2)C;L(-1)C)
Debug.Print Range("A3").Value ' 2,56313865645652
Debug.Print Range("A3").Value2 '2,56313865645652
Debug.Print Range("A3").Text '2,56313866
```

Tout cela est assez cohérent. Vous noterez cependant que dans le cas de la cellule A3 renvoie une valeur différente dans *Value* et dans *Text*. Cela vient du fait que la valeur renvoyée par la propriété *Text* est le contenu **tel qu'il est affiché**.

Prenons maintenant un deuxième exemple. C1 contient la valeur 30/10/2002, C2 contient la formule =TEMPS(10;27;42) et C3 la formule =$C$1+$C$2

Un code de lecture identique au précédent donnerait :

```
Debug.Print Range("C1").Formula '37559
Debug.Print Range("C1").FormulaLocal '37559
Debug.Print Range("C1").FormulaR1C1 '37559
Debug.Print Range("C1").FormulaR1C1Local '37559
Debug.Print Range("C1").Value '30-oct-2002
Debug.Print Range("C1").Value2 '37559
Debug.Print Range("C1").Text '30-10-2002
Debug.Print Range("C2").Formula '=TIME(10,27,42)
Debug.Print Range("C2").FormulaLocal '=TEMPS(10;27;42)
Debug.Print Range("C2").FormulaR1C1 '=TIME(10,27,42)
Debug.Print Range("C2").FormulaR1C1Local '=TEMPS(10;27;42)
Debug.Print Range("C2").Value ' 0,435902777777778
Debug.Print Range("C2").Value2 ' 0,435902777777778
Debug.Print Range("C2").Text '10:27 AM
Debug.Print Range("C3").Formula '=$C$1+$C$2
Debug.Print Range("C3").FormulaLocal '=$C$1+$C$2
Debug.Print Range("C3").FormulaR1C1 '=R1C3+R2C3
Debug.Print Range("C3").FormulaR1C1Local '=L1C3+L2C3
Debug.Print Range("C3").Value '30-oct-2002 10:27:42
Debug.Print Range("C3").Value2 '37559,4359027778
Debug.Print Range("C3").Text '30-10-2002 10:27
```

Détaillons ensemble ces résultats car ils exposent assez bien le fonctionnement de ces propriétés. Nous avons entré la chaîne 30/10/2002. Comme nous l'avons vu précédemment, Excel gère les dates sous la forme d'un nombre entier. Comme il a reconnu une date dans la valeur saisie, il a opéré une conversion de valeur pour transformer la date saisie en entier. Nous l'avons vu en préambule, les propriétés renvoyant les formules renvoient une valeur lorsque la cellule ne contient pas de formule. La valeur réelle de la cellule est l'entier 37559 (nombre de jour entre le 30/10/2002 et le 01/01/1900), toutes les propriétés formules renvoient cet entier. La propriété Value renvoie toujours la valeur formatée. Excel interprète l'entier comme une date, lui applique le formatage par défaut, dans ce cas le format de date courte tel qu'il est défini dans le panneau de configuration du système et renvoie cette valeur dans la propriété Value.

La propriété Value2 renvoie toujours le contenu de la cellule non formaté. Dans ce cas il s'agit donc de l'entier 37559. Text renvoie toujours le contenu de la cellule tel qu'il est affiché. On pourrait s'attendre à ce qu'il renvoie alors la chaîne saisie, mais tel n'est pas le cas. En effet, Excel utilise toujours les séparateurs de dates définit par le système sauf si vous lui avez indiqué explicitement un autre séparateur, soit en modifiant le format de la cellule, soit en modifiant les séparateurs dans la configuration d'Excel. Dans ce cas, vous noterez que la valeur que nous avons saisie n'existe plus (même s'il est assez simple de la reformer).

Pour la cellule C2, vous remarquerez juste que les formules locales utilisent des noms de fonctions localisées (dans notre cas en Français) et le séparateur d'argument localisé (le point virgule) et les propriétés non locales utilisent les noms de fonctions et le séparateur d'argument Américain.

Je viens de dire précédemment que la propriété Value renvoie toujours un contenu formaté, et vous voyez bien que la propriété Value de la cellule C2 renvoie un temps non formaté. N'allez pas pour autant croire que je déraille (encore que…) mais dans ce cas, nous avons utilisé la fonction Temps qui transforme un temps en sa représentation numérique. Excel considérant que nous ne sommes pas suffisamment bourrés pour lui demander de reformater une fonction qui sert justement à ne pas formater sous la forme d'un temps, il renvoie bien dans la propriété Value la valeur renvoyée par la fonction, dans ce cas 0,435902777777778. La propriété Value2 qui elle ne formate jamais renvoie la même valeur. Enfin comme prévu, la propriété Text renvoie la valeur affichée.

Si vous m'avez suivi jusque là, l'interprétation de la cellule C3 ne présente rien de nouveau. Vous noterez juste que la notation américaine R1C1 est sous forme L1C1 dans les versions localisées de la formule.

Prenez deux aspirines, une petite pause, plus qu'un dernier petit col et ensuite c'est tranquille. Vous êtes prêt ? Alors continuons.

A l'exception de la propriété Text, toutes ces propriétés que nous venons de voir présentent donc une autre particularité, ce sont des propriétés tableau. Comme nous l'avons déjà dit, une propriété tableau renvoie ou définit un tableau de valeur quand l'objet Range contient plusieurs cellules.

Autrement dit, ces propriétés renvoient ou définissent un Variant contenant un tableau de Variant représentant le contenu de chaque cellule sous forme de valeurs et de formules. Dans tous les cas, il s'agit d'un tableau à deux dimensions où la première dimension représente les lignes et la seconde les colonnes. La limite basse de chaque dimension vaut toujours 1.

Dans le cas simple des plages continues, le tableau va représenter donc la plage sous la forme :

(1 To Nombre de lignes, 1 To Nombre de colonnes)

Partons d'un fichier exemple qui ressemble à :

L3C9

	1	2	3	4	5	6	7
1	Cellule 21						
2	Date	Heure	T° 1	T° 2	T° 3	T° 4	Moyenne T
3	03/11/2006	14:40:16	40.6	21.5	37.4	40.7	35.05
4	03/11/2006	14:40:17	40.7	21.5	37.5	40.8	35.125
5	03/11/2006	14:40:18	41.2	21.5	37.6	41	35.325
6	03/11/2006	14:40:19	41.1	21.5	37.8	41.2	35.4
7	03/11/2006	14:40:20	41.5	21.5	37.9	41.3	35.55
8	03/11/2006	14:40:21	41.2	21.5	38.1	41.4	35.55
9	03/11/2006	14:40:22	41.8	21.5	38.2	41.6	35.775
10	03/11/2006	14:40:23	41.5	21.5	38.4	41.8	35.8
11	03/11/2006	14:40:24	42.3	21.5	38.4	41.9	36.025
12	03/11/2006	14:40:25	42.4	21.5	38.6	42.1	36.15
13	03/11/2006	14:40:26	42.5	21.5	38.8	42.2	36.25
14	03/11/2006	14:40:27	43	21.5	38.9	42.3	36.425
15	03/11/2006	14:40:28	43.2	21.5	39	42.5	36.55
16	03/11/2006	14:40:29	43.3	21.4	39.2	42.7	36.65
17	03/11/2006	14:40:30	43	21.5	39.3	42.8	36.65
18	03/11/2006	14:40:31	43.4	21.5	39.5	43	36.85
19	03/11/2006	14:40:32	43.7	21.5	39.6	43	36.95
20	03/11/2006	14:40:33	43.7	21.5	39.7	43.2	37.025
21	03/11/2006	14:40:34	43.9	21.5	39.9	43.3	37.15
22	03/11/2006	14:40:35	44	21.5	40	43.4	37.225
23	03/11/2006	14:40:36	44.3	21.5	40.1	43.6	37.375
24	03/11/2006	14:40:37	44.8	21.4	40.2	43.8	37.55
25	03/11/2006	14:40:38	44.7	21.5	40.3	43.9	37.6
26	03/11/2006	14:40:39	44.9	21.5	40.4	44.1	37.725
27	03/11/2006	14:40:40	44.9	21.4	40.6	44.2	37.775
28	03/11/2006	14:40:41	45.3	21.5	40.7	44.3	37.95
29	03/11/2006	14:40:42	45.7	21.5	40.9	44.5	38.15

Nous allons travailler sur la plage L3C3:L30C7 (ou C3:G30 si vous préférez les notations standards).

Fort de ce que nous venons de dire, nous pourrions écrire :

```
Sub Valeurs()

Dim MaPlage As Range, TabValeur As Variant
Dim cmptLigne As Long, NbLigne As Long, NbCol As Long

Set MaPlage = ThisWorkbook.Worksheets("tableau").Range("C3:G30")
TabValeur = MaPlage.Value
NbLigne = UBound(TabValeur, 1)
NbCol = UBound(TabValeur, 2)
MsgBox NbLigne & " lignes" & vbNewLine & NbCol & " colonnes"
For cmptLigne = 1 To NbLigne
    TabValeur(cmptLigne, 1) = TabValeur(cmptLigne, 2) + 1
Next
ThisWorkbook.Worksheets("tableau").Cells(1, 10).Resize(NbLigne,
NbCol).Value = TabValeur

End Sub
```

Dans ce code, je récupère la propriété Value de l'objet Range MaPlage dans une variable TabValeur de type Variant. Comme MaPlage contient plusieurs cellules et que Value est une propriété tableau, TabValeur est un tableau. Je peux donc récupérer le nombre de lignes et de colonnes de ce tableau pour véfifier qu'il fait la même taille que ma plage.

Dans mon tableau, je vais parcourir l'ensemble des lignes de la première colonne et leur affecter comme valeur la valeur de la case située sur la même ligne de la colonne 2 + 1.

Je vais ensuite définir une plage partant de I1 faisant la même taille que mon tableau, et affecter à sa propriété Value la variable TabValeur.

Vous noterez que les cellules de la nouvelle plage contiennent bien les valeurs d'origine sauf pour celle de la première colonne qui ont été modifiées.

Notez aussi que si je n'avais pas de modification à faire, je pourrais affecter directement tel que :

```
Sub Valeurs()

Dim MaPlage As Range, TabValeur As Variant
Dim NbLigne As Long, NbCol As Long

Set MaPlage = ThisWorkbook.Worksheets("tableau").Range("C3:G30")
TabValeur = MaPlage.Value
NbLigne = UBound(MaPlage.Value, 1)
NbCol = UBound(MaPlage.Value, 2)
MsgBox NbLigne & " lignes" & vbNewLine & NbCol & " colonnes"
ThisWorkbook.Worksheets("tableau").Cells(1, 10).Resize(NbLigne,
NbCol).Value = MaPlage.Value

End Sub
```

Mais revenons sur l'exemple précédent. Alors que j'ai modifié la valeur des cellules de la première colonne de la plage, les valeurs de moyenne n'ont pas changées. Si nous allons voir une cellule de la nouvelle colonne moyenne, nous constatons que la formule a disparu et qu'elle ne contient plus que la valeur ce qui n'est pas surprenant puisque nous avons fait une récupération de la propriété Value.

Imaginons un code similaire en travaillant sur les formules. Soit le code :

```
Sub Formule()

Dim MaPlage As Range, TabFormule As Variant
Dim cmptLigne As Long, NbLigne As Long, NbCol As Long

Set MaPlage = ThisWorkbook.Worksheets("tableau").Range("C3:G30")
TabFormule = MaPlage.FormulaLocal
NbLigne = UBound(TabFormule, 1)
NbCol = UBound(TabFormule, 2)
MsgBox NbLigne & " lignes" & vbNewLine & NbCol & " colonnes"
For cmptLigne = 1 To NbLigne
    TabFormule(cmptLigne, 1) = CStr(Val(TabFormule(cmptLigne, 2)) + 1)
Next
ThisWorkbook.Worksheets("tableau").Cells(1, 10).Resize(NbLigne,
NbCol).FormulaLocal = TabFormule

End Sub
```

Notez déjà que nous devons procéder à des conversions de types puisque le tableau récupéré contient des chaînes et non des nombres. Cependant même dans ce cas, la moyenne reste identique à celle de la plage source. Si nous allons voir une cellule de la colonne moyenne, nous voyons que la formule d'origine, `=MOYENNE(LC(-4):LC(-1))` apparaît dans la nouvelle colonne moyenne sous la forme `=MOYENNE(L(2)C(-11):L(2)C(-8))`. C'est-à-dire que la formule s'est modifiée lors de l'affectation à une nouvelle plage pour garder ses arguments de la plage d'origine.

Il convient donc de faire attention, si l'affectation d'un tableau de valeurs équivaut bien à faire un collage spécial des valeurs, la manipulation d'un tableau de formules n'équivaut pas à un collage spécial des formules.

Nous retrouverons ces concepts dans la suite de ce document.

Propriétés de l'objet Range renvoyant un objet Range

Areas (Areas)

La propriété Areas renvoie une collection Areas qui est une collection d'objets Range lorsque l'objet est une plage discontinue. Une plage discontinue contient des éléments qui ne sont pas tous contigus. Prenons un exemple qui sous forme de sélection donnerait :

	A	B	C	D	E	F	G	H
1	Cellule 21							
2	Date	Heure	T° 1	T° 2	T° 3	T° 4	Vitesse	Status
3	03-11-2006	14:40:16	40,6	21,5	37,4	40,7	2497	1
4	03-11-2006	14:40:17	40,7	21,5	37,5	40,8	2500	0
5	03-11-2006	14:40:18	41,2	21,5	37,6	41	2500	1
6	03-11-2006	14:40:19	41,1	21,5	37,8	41,2	2500	1
7	03-11-2006	14:40:20	41,5	21,5	37,9	41,3	2495	1
8	03-11-2006	14:40:21	41,2	21,5	38,1	41,4	2500	0
9	03-11-2006	14:40:22	41,8	21,5	38,2	41,6	2497	0
10	03-11-2006	14:40:23	41,5	21,5	38,4	41,8	2492	1
11	03-11-2006	14:40:24	42,3	21,5	38,4	41,9	2495	0
12	03-11-2006	14:40:25	42,4	21,5	38,6	42,1	2495	0
13	03-11-2006	14:40:26	42,5	21,5	38,8	42,2	2497	0
14	03-11-2006	14:40:27	43	21,5	38,9	42,3	2500	0
15	03-11-2006	14:40:28	43,2	21,5	39	42,5	2500	0
16	03-11-2006	14:40:29	43,3	21,4	39,2	42,7	2500	0
17	03-11-2006	14:40:30	43	21,5	39,3	42,8	2500	0
18	03-11-2006	14:40:31	43,4	21,5	39,5	43	2497	1
19	03-11-2006	14:40:32	43,7	21,5	39,6	43	2500	0
20	03-11-2006	14:40:33	43,7	21,5	39,7	43,2	2497	0
21	03-11-2006	14:40:34	43,9	21,5	39,9	43,3	2500	0
22	03-11-2006	14:40:35	44	21,5	40	43,4	2497	0
23	03-11-2006	14:40:36	44,3	21,5	40,1	43,6	2500	0
24	03-11-2006	14:40:37	44,8	21,4	40,2	43,8	2497	1
25	03-11-2006	14:40:38	44,7	21,5	40,3	43,9	2497	0
26	03-11-2006	14:40:39	44,9	21,5	40,4	44,1	2497	0
27	03-11-2006	14:40:40	44,9	21,4	40,6	44,2	2500	0
28	03-11-2006	14:40:41	45,3	21,5	40,7	44,3	2500	0
29	03-11-2006	14:40:42	45,7	21,5	40,9	44,5	2500	0

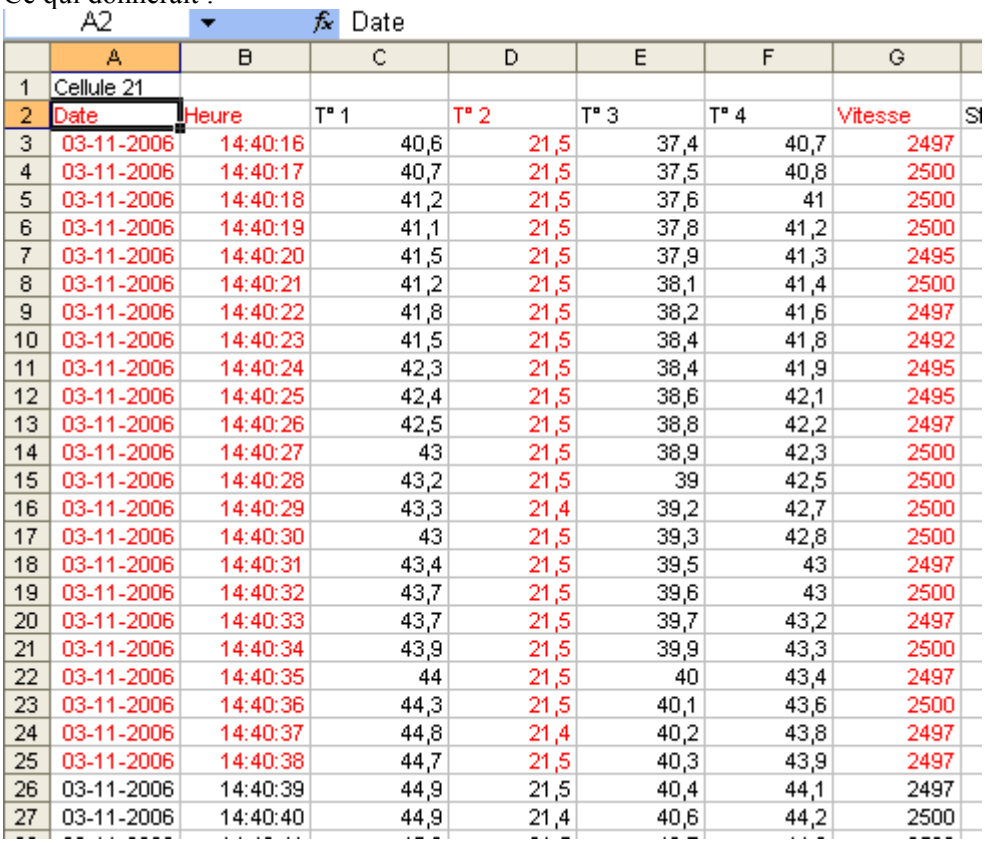
L'objet Range correspondant pourrait être obtenu avec un code tel que :

```
Dim MaPlage As Range

'construction de l'objet range discontinu
With ThisWorkbook.Worksheets("CSF")
    Set MaPlage = .Range(.Cells(2, 1), .Cells(25, 2))
    Set MaPlage = Application.Union(MaPlage, .Cells(2, 4).Resize(24))
    Set MaPlage = Application.Union(MaPlage, .Cells(2, 7).Resize(24))
End With
MaPlage.Select
```

Comme je vous l'ai dit au début de ce chapitre, un objet Range considère toutes les cellules comme une cellule si j'agis sur lui. Je pourrais donc modifier la couleur de la police avec le code

```
MaPlage.Font.Color = vbRed
```

Ce qui donnerait :



	A	B	C	D	E	F	G
1	Cellule 21						
2	Date	Heure	T° 1	T° 2	T° 3	T° 4	Vitesse
3	03-11-2006	14:40:16	40,6	21,5	37,4	40,7	2497
4	03-11-2006	14:40:17	40,7	21,5	37,5	40,8	2500
5	03-11-2006	14:40:18	41,2	21,5	37,6	41	2500
6	03-11-2006	14:40:19	41,1	21,5	37,8	41,2	2500
7	03-11-2006	14:40:20	41,5	21,5	37,9	41,3	2495
8	03-11-2006	14:40:21	41,2	21,5	38,1	41,4	2500
9	03-11-2006	14:40:22	41,8	21,5	38,2	41,6	2497
10	03-11-2006	14:40:23	41,5	21,5	38,4	41,8	2492
11	03-11-2006	14:40:24	42,3	21,5	38,4	41,9	2495
12	03-11-2006	14:40:25	42,4	21,5	38,6	42,1	2495
13	03-11-2006	14:40:26	42,5	21,5	38,8	42,2	2497
14	03-11-2006	14:40:27	43	21,5	38,9	42,3	2500
15	03-11-2006	14:40:28	43,2	21,5	39	42,5	2500
16	03-11-2006	14:40:29	43,3	21,4	39,2	42,7	2500
17	03-11-2006	14:40:30	43	21,5	39,3	42,8	2500
18	03-11-2006	14:40:31	43,4	21,5	39,5	43	2497
19	03-11-2006	14:40:32	43,7	21,5	39,6	43	2500
20	03-11-2006	14:40:33	43,7	21,5	39,7	43,2	2497
21	03-11-2006	14:40:34	43,9	21,5	39,9	43,3	2500
22	03-11-2006	14:40:35	44	21,5	40	43,4	2497
23	03-11-2006	14:40:36	44,3	21,5	40,1	43,6	2500
24	03-11-2006	14:40:37	44,8	21,4	40,2	43,8	2497
25	03-11-2006	14:40:38	44,7	21,5	40,3	43,9	2497
26	03-11-2006	14:40:39	44,9	21,5	40,4	44,1	2497
27	03-11-2006	14:40:40	44,9	21,4	40,6	44,2	2500

Imaginons maintenant que nous souhaitions récupérer le nombre de cellules et le nombre de colonnes de l'objet Range.

```
Sub TestArea()

    Dim MaPlage As Range

    'construction de l'objet range discontinu
    With ThisWorkbook.Worksheets("CSF")
        Set MaPlage = .Range(.Cells(2, 1), .Cells(25, 2))
        Set MaPlage = Application.Union(MaPlage, .Cells(2, 4).Resize(24))
        Set MaPlage = Application.Union(MaPlage, .Cells(2, 7).Resize(24))
    End With
    MsgBox MaPlage.Cells.Count '96
    MsgBox MaPlage.Columns.Count '2 ????

End Sub
```

Si le compte de cellules est juste, le compte de colonnes, lui, ne l'est pas. De fait, la propriété Columns ne sait pas répondre sur une plage discontinue et renvoie dans cet exemple le nombre de colonnes **de la première plage continu** contenue dans la plage discontinue. La notion de première plage s'entend dans le sens des références Excel, c'est-à-dire la plus à gauche et/ou la plus haute.

Pour avoir le bon compte de colonnes, il faudrait écrire un code comme :

```
    Dim cmptCol As Integer, tmpPlage As Range
    If MaPlage.Areas.Count > 1 Then
        For Each tmpPlage In MaPlage.Areas
            cmptCol = cmptCol + tmpPlage.Columns.Count
        Next
    Else
    cmptCol = MaPlage.Columns.Count
    End If
    MsgBox cmptCol
```

Notez que cette propriété est intéressante sur le fait qu'une simple évaluation de sa propriété Count permet de savoir si la plage représentée par l'objet Range est continue ou non.

Cells (Range)

Eh oui, c'est le cercle maudit. Un objet Range expose une propriété Cells qui renvoie la collection des cellules sous la forme d'un objet Range qui renvoie une collection Cells (et ainsi de suite).

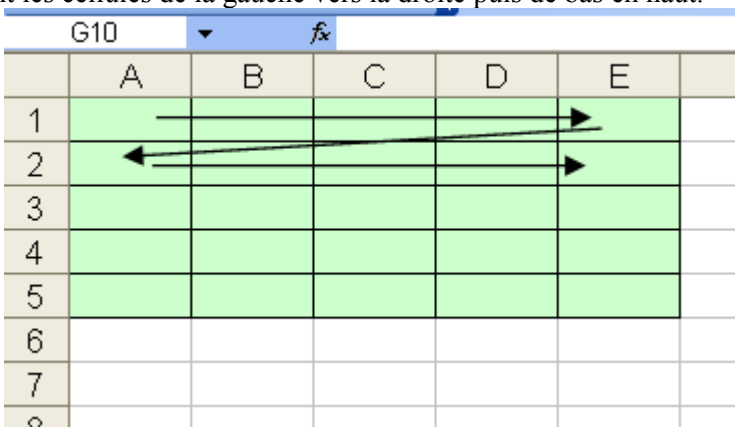
A part vous embrouiller on utilise cette propriété explicitement soit pour désigner une cellule de la plage, soit pour énumérer les cellules de la plage.

Pour l'énumération pas de problème.

```
Sub EnumCell()

    Dim MaCell As Range, Somme As Double

    For Each MaCell In ThisWorkbook.Worksheets("CSF").Range("C3:C30").Cells
        If IsNumeric(MaCell.Value) Then Somme = Somme + MaCell.Value
    Next MaCell
    MsgBox Somme

End Sub
```

Pour la désignation, c'est déjà autrement périlleux. Raisonnons sur la plage "A1 : E5".

Les cellules de cette plage peuvent être désignées soit par leur position absolue dans la plage, soit par leur position "linéaire", c'est-à-dire comme si la plage était sur une colonne ou sur une ligne en lisant les cellules de la gauche vers la droite puis de bas en haut.



Jusque là rien de bien sorcier, on peut écrire :

```
Sub Designation()

    Dim Maplage As Range
    Set Maplage = ThisWorkbook.Worksheets("Feuil1").Range("A1:E5")
    'désignation absolu
    Debug.Print Maplage.Cells(3, 2).Address(False, False) 'B3
    'désignation linéaire
    Debug.Print Maplage.Cells(13).Address(False, False) 'C3

End Sub
```

Malheureusement on peut aussi écrire

```
Sub Designation()

    Dim Maplage As Range
    Set Maplage = ThisWorkbook.Worksheets("Feuil1").Range("A1:E5")
    'désignation absolu
    Debug.Print Maplage.Cells(6, 2).Address(False, False) 'B6
    'désignation linéaire
    Debug.Print Maplage.Cells(29).Address(False, False) 'D6

End Sub
```

Et atteindre ainsi des cellules qui ne sont pas dans la plage.

De plus il faut faire attention car la désignation dite absolue est en fait relative à la plage, donc on aurait :

```
Sub Designation()

    Dim Maplage As Range

    Set Maplage = ThisWorkbook.Worksheets("Feuil1").Range("B2:F6")
    Debug.Print Maplage.Cells(6, 2).Address(False, False) 'C7
    Debug.Print Maplage.Cells(0, 0).Address(False, False) 'A1

End Sub
```

Donc méfiez vous de la désignation en partant d'une plage et essayez plutôt de travailler sur des références fixes.

Columns & Rows

Si la manipulation de ces propriétés en partant de la feuille est assez triviale, elle peut être un peu plus piégeuse lorsqu'on travaille sur des objets Range.

Pour les plages continues, ces propriétés renvoient un objet Range contenant toutes les cellules appartenant à la colonne ou à la ligne de l'objet Range d'où elles sont appelées. Un petit exemple est plus simple à comprendre.

```
Sub TestColRow()

    Dim MaPlage As Range

    Set MaPlage = ThisWorkbook.Worksheets("tableau").Range("C3:G30")
    Debug.Print MaPlage.Columns(2).Address(False, False, xlA1)
    'D3:D30
    Debug.Print MaPlage.Rows(2).Address(False, False, xlA1)
    'C4:G4

End Sub
```

Autrement dit, c'est un objet Range représentant l'intersection entre la ligne ou la colonne et l'objet Range. Attention, si la colonne ou la ligne n'appartiennent pas à l'objet Range, c'est une plage décalée qui va être renvoyée. Celle-ci sera alors l'intersection entre par exemple la ligne spécifiée et les colonnes de l'objet Range appelant. Par exemple :

```
    Debug.Print MaPlage.Rows(31).Address(False, False, xlA1)
    'C33:G33
```

Lorsqu'on souhaite obtenir une plage de colonnes ou de lignes continues, on passe un argument sous forme de texte précisant les colonnes (en lettres) ou les lignes (en chiffres). Attention, il s'agit d'une référence relative, c'est-à-dire que pour un objet Range "C3:G30", l'appel de la propriété Columns("B") renvoie "D3:D30".

Comme ces propriétés renvoient un objet Range, il est possible de les enchaîner :

```
    Debug.Print MaPlage.Columns("A:C").Rows("2:8").Address(False, False,
xlA1)
    'C4:E10
```

Dependents, DirectDependents, Precedents & DirectPrecedents

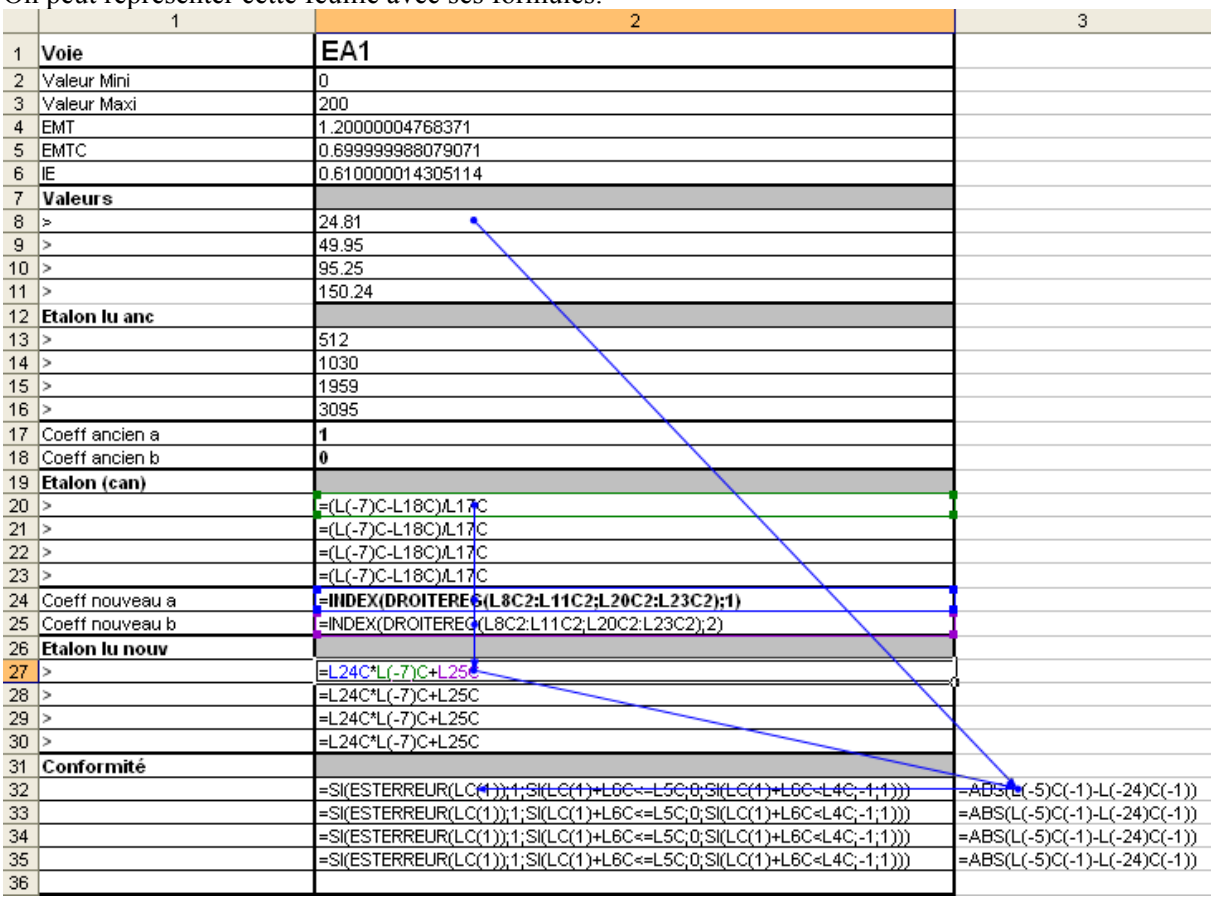
Ces propriétés sont assez complexes à manipuler. Elles sont issues de la notion de plages liées, c'est-à-dire de cellules étant des arguments de formule d'autres cellules. Par définition, une cellule dépendante est une cellule contenant une formule utilisant une autre cellule, une cellule précédente est une cellule dont la valeur est utilisée par la formule d'une autre cellule.

Les cellules sont directement précédentes ou dépendantes si leur relation est directe.

Pour bien comprendre ces principes, nous allons prendre comme exemple une feuille de dépouillement d'un étalonnage de voie d'acquisition.

	1	2	3	4
1	**Voie**	**EA1**		
2	Valeur Mini	0		
3	Valeur Maxi	200		
4	EMT	1.200000048		
5	EMTC	0.699999988		
6	IE	0.610000014		
7	**Valeurs**			
8	>	24.81		
9	>	49.95		
10	>	95.25		
11	>	150.24		
12	**Etalon lu anc**			
13	>	512		
14	>	1030		
15	>	1959		
16	>	3095		
17	Coeff ancien a	**1**		
18	Coeff ancien b	**0**		
19	**Etalon (can)**			
20	>	512		
21	>	1030		
22	>	1959		
23	>	3095		
24	Coeff nouveau a	**0.048576014**		
25	Coeff nouveau b	-0.039347523		
26	**Etalon lu nouv**			
27	>	24.83157178		
28	>	49.99394717		
29	>	95.12106442		
30	>	150.3034166		
31	**Conformité**			
32		Conforme		
33		Conforme		
34		conforme avec risques		
35		Conforme		
36				

On peut représenter cette feuille avec ses formules.



	1	2	3
1	**Voie**	**EA1**	
2	Valeur Mini	0	
3	Valeur Maxi	200	
4	EMT	1.20000004768371	
5	EMTC	0.699999988079071	
6	IE	0.610000014305114	
7	**Valeurs**		
8	>	24.81	
9	>	49.95	
10	>	95.25	
11	>	150.24	
12	**Etalon lu anc**		
13	>	512	
14	>	1030	
15	>	1959	
16	>	3095	
17	Coeff ancien a	**1**	
18	Coeff ancien b	**0**	
19	**Etalon (can)**		
20	>	=(L(-7)C-L18C)/L17C	
21	>	=(L(-7)C-L18C)/L17C	
22	>	=(L(-7)C-L18C)/L17C	
23	>	=(L(-7)C-L18C)/L17C	
24	Coeff nouveau a	**=INDEX(DROITEREG(L8C2:L11C2;L20C2:L23C2);1)**	
25	Coeff nouveau b	=INDEX(DROITEREG(L8C2:L11C2;L20C2:L23C2);2)	
26	**Etalon lu nouv**		
27	>	=L24C*L(-7)C+L25C	
28	>	=L24C*L(-7)C+L25C	
29	>	=L24C*L(-7)C+L25C	
30	>	=L24C*L(-7)C+L25C	
31	**Conformité**		
32		=SI(ESTERREUR(LC(1));1;SI(LC(1)+L6C<=L5C;0;SI(LC(1)+L6C<L4C;-1;1)))	=ABS(L(-5)C(-1)-L(-24)C(-1))
33		=SI(ESTERREUR(LC(1));1;SI(LC(1)+L6C<=L5C;0;SI(LC(1)+L6C<L4C;-1;1)))	=ABS(L(-5)C(-1)-L(-24)C(-1))
34		=SI(ESTERREUR(LC(1));1;SI(LC(1)+L6C<=L5C;0;SI(LC(1)+L6C<L4C;-1;1)))	=ABS(L(-5)C(-1)-L(-24)C(-1))
35		=SI(ESTERREUR(LC(1));1;SI(LC(1)+L6C<=L5C;0;SI(LC(1)+L6C<L4C;-1;1)))	=ABS(L(-5)C(-1)-L(-24)C(-1))
36			

Les flèches représentent les dépendances telles qu'elles sont affichées par la barre d'audit. Nous allons visualiser ces dépendances par le code en modifiant le fond des cellules avec un code VBA, en prenant la cellule L32C3 (C32) comme cellule de référence.

Nous allons donc mettre en jaune les cellules précédentes, en rouge les cellules directement précédentes et en bleu clair les cellules dépendantes.

```
Sub AntecedentDependent()

    Dim MaPlage As Range

    On Error Resume Next
    Set MaPlage = ThisWorkbook.Worksheets("Depend").Cells(32, 4)
    MaPlage.Precedents.Interior.Color = vbYellow
    MaPlage.DirectPrecedents.Interior.Color = vbRed
    MaPlage.Dependents.Interior.Color = vbCyan
    On Error Goto 0

End Sub
```

Notez que je suis en mode "No Kill", c'est-à-dire que je me suis mis en mode de traitement d'erreurs immédiat sans pour autant traiter les erreurs. En effet, VBA va lever une erreur 1004 s'il n'y a pas de cellules antécédentes ou précédentes. En l'occurrence, il n'y a pas d'incidence à ne pas traiter l'erreur puisque nous cherchons juste à colorier des cellules le cas échéant.

Ce code nous donnera un résultat tel que :

	1	2	3
1	**Voie**	**EA1**	
2	Valeur Mini	0	
3	Valeur Maxi	200	
4	EMT	1.200000048	
5	EMTC	0.699999988	
6	IE	0.610000014	
7	**Valeurs**		
8	>	24.81	
9	>	49.95	
10	>	95.25	
11	>	150.24	
12	**Etalon lu anc**		
13	>	512	
14	>	1030	
15	>	1959	
16	>	3095	
17	Coeff ancien a	**1**	
18	Coeff ancien b	**0**	
19	**Etalon (can)**		
20	>	512	
21	>	1030	
22	>	1959	
23	>	3095	
24	Coeff nouveau a	**0.048576014**	
25	Coeff nouveau b	-0.039347523	
26	**Etalon lu nouv**		
27	>	24.83157178	
28	>	49.99394717	
29	>	95.12106442	
30	>	150.3034166	
31	**Conformité**		
32		Conforme	0.021571781
33		Conforme	0.04394717
34		conforme avec risques	0.128935578
35		Conforme	0.063416627
36			

Vous remarquerez qu'il faut évidemment colorier les cellules précédentes avant les cellules directement précédentes si vous voulez faire la différence entre les deux puisque par définition, l'objet renvoyé par la collection des cellules précédentes contient les cellules directement précédentes. Il en serait de même pour les cellules dépendantes.

Les cellules précédentes et dépendantes ne sont pas souvent manipulées. De fait, seul certains scénarii particuliers requièrent ce genre de programmation, le cas le plus connu étant la gestion contrôlée des calculs dans les feuilles denses.

Par exemple, le code suivant recalcule uniquement les cellules dépendantes d'une cellule modifiée.

```
Private Sub Worksheet_Change(ByVal Target As Range)

Static EnCours As Boolean

    If EnCours Then Exit Sub
    EnCours = True
    On Error Resume Next
    Target.DirectDependents.Calculate
    Do While Err.Number = 0
        Set Target = Target.DirectDependents
        Target.DirectDependents.Calculate
    Loop
    Err.Clear
    On Error GoTo 0
    EnCours = False

End Sub
```

End

La propriété End renvoie la cellule de fin de zone d'une plage donnée. La notion de zone est assez simple à comprendre et assez complexe à employer. Une zone pour Excel, c'est une plage linéaire de cellules contenant des valeurs ou n'en contenant pas. Prenons la feuille exemple suivante :

	A	B	C	D	E	F	G	H
1	Date	Heure	T° 1	T° 2	T° 3	T° 4	Moyenne	
2	03-11-2006	14:40:16	40.6	21.5	37.4	40.7		
3	03-11-2006	14:40:17	40.7	21.5		40.8		
4	03-11-2006	14:40:18	41.2	21.5	37.6	41		
5	03-11-2006	14:40:19	41.1	21.5	37.8	41.2		
6	03-11-2006	14:40:20	41.5	21.5	37.9			
7	03-11-2006	14:40:21	41.2	21.5	38.1	41.4		
8	03-11-2006	14:40:22	41.8	21.5	38.2	41.6		
9	03-11-2006	14:40:23	41.5	21.5	38.4	41.8		
10	03-11-2006	14:40:24	42.3	21.5		41.9		
11	03-11-2006	14:40:25	42.4	21.5	38.6	42.1		
12	03-11-2006	14:40:26	42.5	21.5	38.8			
13	03-11-2006	14:40:27	43	21.5	38.9	42.3		
14	03-11-2006	14:40:28	43.2	21.5	39	42.5		
15	03-11-2006	14:40:29	43.3	21.4	39.2	42.7		
16	03-11-2006	14:40:30	43	21.5		42.8		
17	03-11-2006	14:40:31	43.4	21.5		43		
18	03-11-2006	14:40:32	43.7	21.5		43		
19	03-11-2006	14:40:33	43.7	21.5	39.7	43.2		
20	03-11-2006	14:40:34	43.9	21.5	39.9	43.3		
21	03-11-2006	14:40:35	44	21.5	40	43.4		
22	03-11-2006	14:40:36	44.3	21.5	40.1	43.6		
23	03-11-2006	14:40:37	44.8	21.4	40.2	43.8		
24	03-11-2006	14:40:38		21.5	40.3	43.9		
25	03-11-2006	14:40:39		21.5	40.4	44.1		
26	03-11-2006	14:40:40		21.4	40.6			
27	03-11-2006	14:40:41		21.5	40.7	44.3		
28	03-11-2006	14:40:42		21.5	40.9	44.5		
29	03-11-2006	14:40:43		21.4	41.1	44.7		
30								

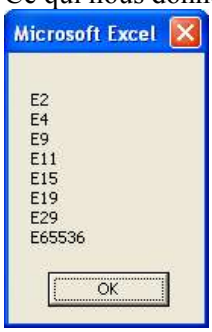
Commençons par des cas simples

```
Sub TestEnd()

    Dim MaPlage As Range

    Set MaPlage = ThisWorkbook.Worksheets("End").Range("A1")
    Debug.Print MaPlage.End(xlToRight).Address(False, False, xlA1)
    'G1
    Debug.Print MaPlage.End(xlDown).Address(False, False, xlA1)
    'A29
    Set MaPlage = ThisWorkbook.Worksheets("End").Range("C2")
    Debug.Print MaPlage.End(xlToRight).Address(False, False, xlA1)
    'F2
    Debug.Print MaPlage.End(xlDown).Address(False, False, xlA1)
    'C23
    Set MaPlage = ThisWorkbook.Worksheets("End").Range("E1")
    Debug.Print MaPlage.End(xlToRight).Address(False, False, xlA1)
    'G1
    Debug.Print MaPlage.End(xlDown).Address(False, False, xlA1)
    'E2
    Set MaPlage = ThisWorkbook.Worksheets("End").Range("G1")
    Debug.Print MaPlage.End(xlToRight).Address(False, False, xlA1)
    'IV1
    Debug.Print MaPlage.End(xlDown).Address(False, False, xlA1)
    'G65536

End Sub
```

Ces exemples renvoient la cellule de fin de zone. Dans ce cas je fais la recherche en bas ou à droite, mais c'est équivalent en haut ou à gauche. Regardons les deux derniers exemples, lorsque G1 est la cellule de base. Dans les deux cas, c'est la dernière cellule de la feuille dans la direction de recherche qui est renvoyée. En effet, End ne renvoie jamais la cellule appelante comme cellule de fin de zone, dans ce cas, End cherche la fin de la zone suivante, la fin de la feuille en l'occurrence puisqu'il n'y a plus que des cellules vides. Pour bien comprendre ce fonctionnement, imaginons le code suivant :

```
Sub TestEnd1()

    Dim MaPlage As Range, Message As String

    Set MaPlage = ThisWorkbook.Worksheets("End").Range("E1")
    Do
        Set MaPlage = MaPlage.End(xlDown)
        Message = Message & vbCrLf & MaPlage.Address(False, False, xlA1)
    Loop Until MaPlage.Row = 65536
    MsgBox Message

End Sub
```

Ce qui nous donnera :



Comme vous le voyez, les cellules renvoyées contiennent toujours une valeur à l'exception de la dernière, la recherche de fin de zone est donc bien basée sur les cellules ayant une valeur. Cette méthode End est souvent utilisée pour renvoyer des plages de valeurs continues sans avoir à connaître au préalable le nombre de cellule de la plage. Par exemple le code suivant renvoie la première colonne de valeur de la feuille :

```
    With ThisWorkbook.Worksheets("End")
        Set MaPlage = .Range(.Cells(1, 1), .Cells(1, 1).End(xlDown))
        'on trouve parfois la notation
        'Set MaPlage = .Range("A1", .Range("A1").End(xlDown))
    End With
```

La recherche de la dernière cellule pour les développeurs VBA c'est un peu comme le Saint Graal pour les chevaliers de la table ronde. Je vous exposerai le problème et les solutions dans l'étude des techniques classiques.

EntireRow & EntireColumn

Renvoie la ou les colonnes (lignes) entières de la plage appelante.

```
Sub TestEntire()

    Dim MaPlage As Range

    Set MaPlage = ThisWorkbook.Worksheets("Tableau").Range("J1:N28")
    Debug.Print MaPlage.EntireColumn.Address(False, False, xlA1)
    'J:N
    Debug.Print MaPlage.EntireRow.Address(False, False, xlA1)
    '1:28

End Sub
```

MergeArea

S'applique généralement sur une cellule. Renvoie la plage fusionnée si la cellule appartient à une plage fusionnée ou la cellule si tel n'est pas le cas. Ne confondez pas cette propriété qui renvoie une plage avec la propriété MergedCells qui renvoie vraie si la cellule fait partie d'une plage fusionnée.

Offset

Renvoie une plage décalée par rapport à la plage appelante. De la forme :
Property Offset(*[RowOffset As Long], [ColumnOffset As Integer]*) **As Range**
Où *RowOffset* est un entier long définissant le décalage de ligne et *ColumnOffset* est un entier définissant le décalage de colonne. Ces arguments suivent les règles :

- Si la valeur de l'argument est négative, le décalage aura lieu vers le haut pour les lignes et vers la gauche pour les colonnes.
- Si l'argument est positif, le décalage aura lieu vers le bas pour les lignes et vers la droite pour les colonnes.
- Si l'argument est nul ou omis il n'y aura pas de décalage
- Si le décalage demandé renvoie une ligne ou une colonne en dehors de l'intervalle de la feuille (1 à 65536 pour les lignes ; 1 à 256 pour les colonnes), une erreur sera levée.

La plage renvoyée a le même nombre de lignes / colonnes que la plage appelante.
Le code suivant efface les cellules contenant la valeur 0 dans la plage "A2:G2000" de la feuille tableau.

```
Sub TestOffset()
    Dim MaPlage As Range, cmptCol As Long, cmptLig As Long

    Set MaPlage = ThisWorkbook.Worksheets("Tableau").Cells(2, 1)
    For cmptLig = 0 To MaPlage.End(xlDown).Row - MaPlage.Row
        For cmptCol = 0 To MaPlage.End(xlToRight).Column - MaPlage.Column
            If MaPlage.Offset(cmptLig, cmptCol).Value = 0 Then
MaPlage.ClearContents
        Next cmptCol
    Next cmptLig
End Sub
```

Resize

Redimensionne un objet Range du nombre de colonnes / lignes passé en argument. De la forme:
Property Resize(*[RowSize As Long], [ColumnSize As Integer]*) **As Range**
Où *RowSize* est un entier long définissant le nombre de lignes et *ColumnSize* est un entier définissant le nombre de colonnes. Ces arguments sont forcément des valeurs positives ou sont omis, dans ce cas la plage n'est pas redimensionnée. Si la plage renvoyée n'est pas dans les limites admissibles de la feuille, une erreur sera levée.
En combinant cette propriété et la propriété Offset vue précédemment, il est possible de redéfinir n'importe quelle plage sur la feuille en partant d'une autre plage. Cette façon de travailler peut alors remplacer l'utilisation de plage fixe comme nous le verrons dans la discussion technique plus loin.
L'exemple suivant rempli la colonne moyenne de la feuille tableau.

```
Sub TestResize()
    Dim MaPlage As Range, cmptLig As Long

    Set MaPlage = ThisWorkbook.Worksheets("Tableau").Cells(2, 1)
    MaPlage.Offset(, 6).Value = "Moyenne T1-T4"
    'parcours la plage en ligne
    For cmptLig = 1 To MaPlage.End(xlDown).Row - MaPlage.Row
        MaPlage.Offset(cmptLig, 6).FormulaLocal = "=Moyenne(" &
MaPlage.Offset(cmptLig, 2).Resize(, 4).AddressLocal(True, True, xlR1C1) &
")"
    Next cmptLig
End Sub
```

Autres Propriétés de l'objet Range

Address & AddressLocal (String)

Renvoie l'adresse de la plage.

Property Address(*[RowAbsolute As Boolean], [ColumnAbsolute As Boolean], [ReferenceStyle As XlReferenceStyle = xlA1], [External As Boolean], [RelativeTo As Range]*) **As String**

Les arguments *RowAbsolute* et *ColumnAbsolute* détermine si l'adresse renvoyée est absolue ou relative, *ReferenceStyle* vaut xlA1 ou xlR1C1. Si *External* est vrai l'adresse contiendra le nom du classeur et de la feuille, *RelativeTo* précise la plage ou la cellule de référence pour les références relatives. La propriété AddressLocal fonctionne à l'identique mais les références de type R1C1 sont converties en L1C1.

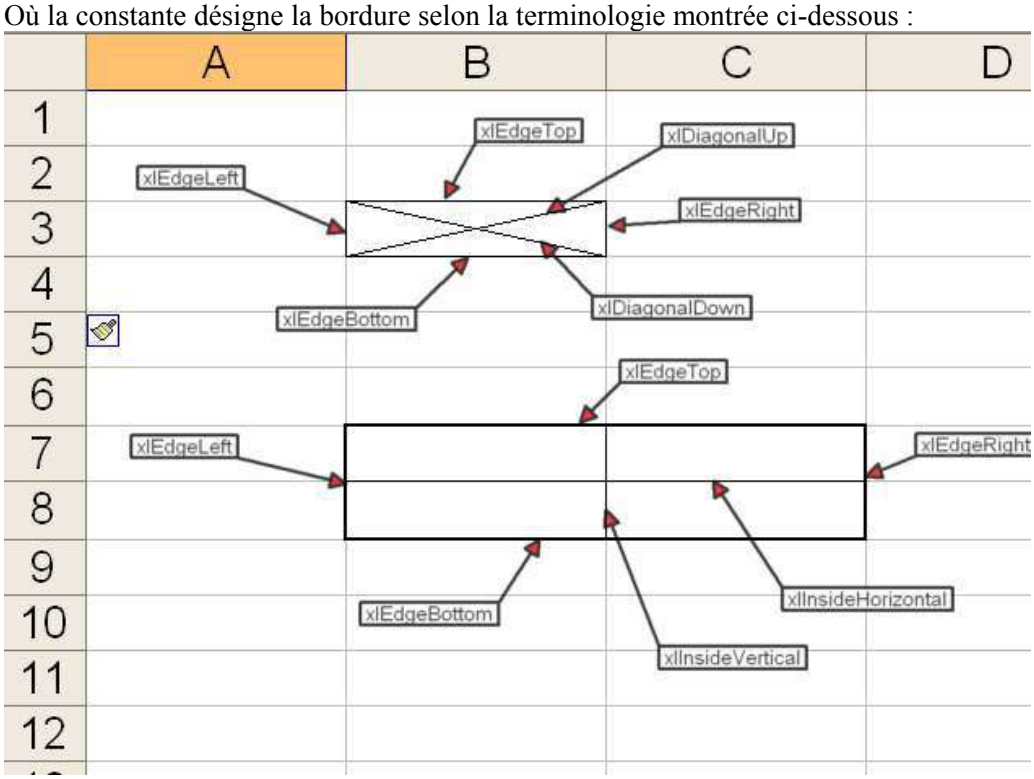
```
Sub TestAdresse()

    Dim MaPlage As Range
    Set MaPlage = ThisWorkbook.Worksheets("Feuil1").Range("A1:A10")
    Debug.Print MaPlage.AddressLocal(True, True, xlR1C1)
    'L1C1:L10C1
    Set MaPlage = ThisWorkbook.Worksheets("Feuil1").Cells(5, 6)
    Debug.Print MaPlage.Address(False, False, xlR1C1, False,
ThisWorkbook.Worksheets("Feuil1").Range("B2"))
    'R[3]C[4]
    Set MaPlage = Application.Union(MaPlage,
ThisWorkbook.Worksheets("Feuil1").Range("A1:A10"))
    Debug.Print MaPlage.Address
    '$F$5,$A$1:$A$10

End Sub
```

Borders (Borders)

Renvoie la collection des bordures de la plage. Les bordures sont différentes selon que la plage contienne une ou plusieurs cellules. Une bordure spécifique est donc désignée avec la notation :

```
Range.Borders(Constante)
```

Où la constante désigne la bordure selon la terminologie montrée ci-dessous :



Une bordure est définie selon trois propriétés :

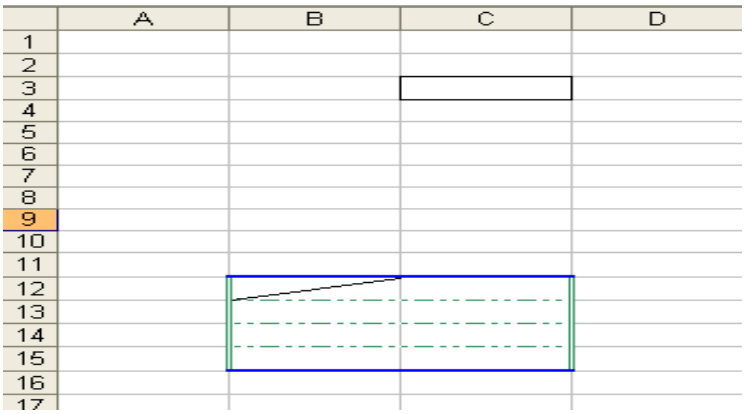
- Color (ou ColoIindex) qui définit la couleur du trait
- LineStyle qui définit le style du trait
- Weight qui définit son épaisseur.

Si vous appelez la propriété Borders sans préciser de constante, ce sont les quatre bordures du tour extérieur qui seront affectés.

Les encadrements complexes engendrent des codes assez lourds.

```
Sub Encadrement()

    Dim MaFeuille As Worksheet

    Set MaFeuille = ThisWorkbook.Worksheets("Feuil1")
    MaFeuille.Cells(3, 3).Borders.LineStyle = xlContinuous
    With MaFeuille.Range("B12:C15")
        With .Borders(xlEdgeLeft)
            .LineStyle = xlDouble
            .Weight = xlThick
            .ColorIndex = 50
        End With
        With .Borders(xlEdgeTop)
            .LineStyle = xlContinuous
            .Weight = xlMedium
            .ColorIndex = 5
        End With
        With .Borders(xlEdgeBottom)
            .LineStyle = xlContinuous
            .Weight = xlMedium
            .ColorIndex = 5
        End With
        With .Borders(xlEdgeRight)
            .LineStyle = xlDouble
            .Weight = xlThick
            .ColorIndex = 50
        End With
        With .Borders(xlInsideHorizontal)
            .LineStyle = xlDashDotDot
            .Weight = xlThin
        End With
    End With
End Sub
```

Ce code donnera :



Characters (Characters)

Bien qu'on l'utilise assez peu, cette propriété renvoie le contenu de la cellule sous forme d'une collection de caractères, ce qui permet de manipuler ceux-ci séparément, d'insérer une chaîne ou de supprimer certains caractères. Généralement, on utilise plutôt des fonctions VBA de traitement de chaînes, mais dans certains cas, la collection Characters est la seule solution, par exemple pour utiliser des polices différentes dans une même cellule ou pour mettre en couleur certains caractères.

Column & Row (long)

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

ColumnWidth & RowHeight (Double)

Renvoie la hauteur de ligne ou la largeur de la colonne.
Plage hétérogène ⇨ Null
Attention, la hauteur de ligne renvoie des points alors que la largeur de colonne renvoie un équivalent nombre de caractères. Si vous voulez connaître la largeur en nombre de points, vous devez utiliser la propriété Width

Font (Font)

Renvoie un objet Font qui est la police utilisée pour la plage.
Plage hétérogène ⇨ Font générique
Si toutes les cellules contiennent la même police, la propriété renvoie cette police, sinon elle renvoie un objet Font générique dont le nom est Null.
Le code suivant vérifie que toutes les cellules de la plage ont la même police, si tel n'est pas le cas, il appliquera une police Arial 11 Gras + Bleu à toute la plage.

```
Sub TestFont()

    Dim MaPlage As Range, Police As Font

    Set MaPlage = ThisWorkbook.Worksheets("Tableau").Range("P1:P10")
    Debug.Print MaPlage.Font.Name
    Set Police = MaPlage.Font
    If IsNull(Police.Name) Then
        With Police
            .Name = "Arial"
            .Color = vbBlue
            .Bold = True
            .Size = 11
        End With
    End If

End Sub
```

HasFormula (Boolean)

Renvoie vrai si toutes les cellules de la plage contiennent une valeur, Faux si aucune cellule n'en contient, Null si certaines cellules en contiennent.

Hidden (Boolean)

Cette propriété s'applique uniquement pour des plages contenant des lignes ou des colonnes entières. Renvoie vrai si toutes les lignes ou toutes les colonnes de la plage sont masquées, Faux sinon.
Pour savoir si une cellule est masquée on utilise donc :
Cellule.EntireRow.Hidden ou Cellule.EntireColumn.Hidden

HorizontalAlignment & VerticalAlignment (Variant)

Renvoie ou définit l'alignement dans les cellules.
Plage hétérogène ⇨ Null
Les valeurs de centrage sont les suivantes :

HorizontalAlignment		
Constante	**Valeur**	**Commentaire**
XILeft	-4131	Gauche
xICenter	-4108	Centré
xIRight	-4152	Droite
xIFill	5	Recopié
xIJustify	-4130	Justifié
xlCenterAcrossSelection	7	Centré sur plusieurs colonnes
VerticalAlignment		
xITop	-4160	Haut
xICenter	-4108	Centré
xIBottom	-4107	Bas
XIJustify	-4130	Justifié
XIDistributed	-4117	Distribué

Interior (Interior)

Renvoie un objet Interior qui est le format du corps de la cellule.
Plage hétérogène ⇨ Interior générique
L'objet Interior gère la couleur et le motif du fond de la cellule.

Left & Top (Single)

Renvoie la position en point du bord gauche ou du sommet de la plage. Ces propriétés sont évidemment en lecture seule. On ne les utilise généralement que lorsqu'on souhaite insérer un objet dans une feuille à une position de cellule particulière.

```
With ThisWorkbook.Worksheets("Tableau")
    .ChartObjects.Add .Range("P4").Left, .Range("P4").Top, 500, 200
End With
```

Locked (Boolean)

Renvoie vrai si toutes les cellules de la plage sont verrouillées, Faux si aucune ne l'est. Attention le verrouillage ne prend effet que lorsque la feuille est protégée. Ceci veut dire que la cellule peut être verrouillée au sens de la propriété Locked sans l'être physiquement si la feuille n'est pas protégée.
Plage hétérogène ⇨ Null

MergeCells (Boolean)

Renvoie vrai si toutes les cellules de la plage sont fusionnées, Faux si aucune ne l'est. Les cellules peuvent être toutes fusionnées dans la même plage ou dans des plages différentes.
Plage hétérogène ⇨ Null

Name (String)

Renvoie ou définit le nom de la plage. Par défaut le nom sera ajouté à la collection Names du classeur. Si le nom existait déjà dans le classeur, son ancienne référence sera écrasée.

Pour attribuer le nom à la collection Names de la feuille, vous devez définir un nom (chaîne de caractères) de la forme "NomFeuille!Nom"

```
Sub TestName()

Dim MaPlage As Range

    Set MaPlage = ThisWorkbook.Worksheets("Tableau").Range("J1:N28")
    'Ajoute Nom1 à la collection du classeur en écrasant l'ancienne
référence
    MaPlage.Name = "Nom1"
    'Ajoute Nom2 à la collection Names de la feuille nommée Tableau
    MaPlage.Name = "Tableau!Nom2"

End Sub
```

NumberFormat & NumberFormatLocal (String)

Renvoie la chaîne de format des cellules de la plage.

Plage hétérogène ⇨ Null

La valeur renvoyée peut être une chaîne prédéfinie ou une chaîne personnalisée. Elle n'est renvoyée que si toutes les cellules de la plage ont la même chaîne de formatage.

Orientation (Integer)

Renvoie ou définti l'orientation du texte dans la cellule. Valeur entière comprise entre -90 et 90 degrés.

Plage hétérogène ⇨ Null

Les textes normaux (sans orientation définie) renvoient -4128 (xlHorizontal)

Style (Variant)

Renvoie ou définit le nom du style pour la plage de cellules.

Plage hétérogène ⇨ Nothing

Méthodes de l'objet Range

AddComment

Ajoute un commentaire à la cellule. Cette méthode lève une exception si la plage contient plusieurs cellules ou si la cellule contient déjà un commentaire. De la forme :

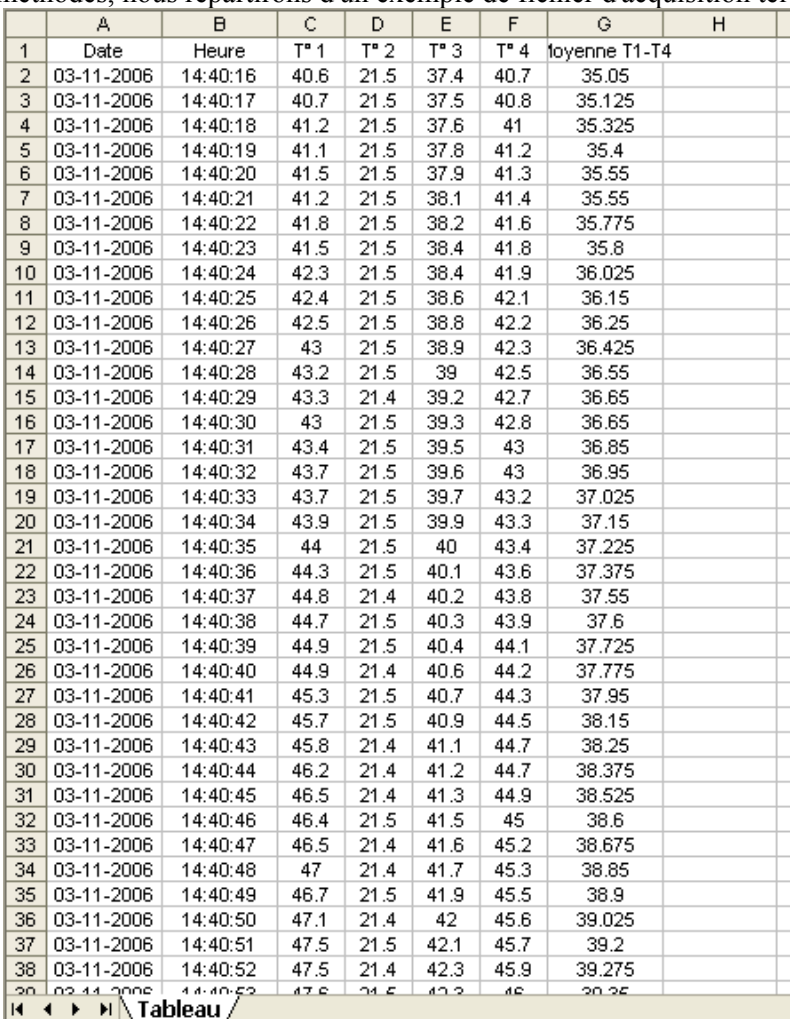
Function AddComment(*[Text As String]*) **As Comment**

Où *Text* est le texte du commentaire à ajouter.

```
Sub TestMethod()

Dim MaPlage As Range, Commentaire As Comment

    Set MaPlage = ThisWorkbook.Worksheets("Methode").Range("A1")
    Set Commentaire = MaPlage.AddComment("Commentaire")
    Commentaire.Shape.AutoShapeType = msoShapeCross

End Sub
```

AutoFilter

Cette méthode est extrêmement puissante dans les scénarii de recherche de valeur. Pour illustrer ces méthodes, nous repartirons d'un exemple de fichier d'acquisition tel que :



	A	B	C	D	E	F	G	H
1	Date	Heure	T° 1	T° 2	T° 3	T° 4	Moyenne T1-T4	
2	03-11-2006	14:40:16	40.6	21.5	37.4	40.7	35.05	
3	03-11-2006	14:40:17	40.7	21.5	37.5	40.8	35.125	
4	03-11-2006	14:40:18	41.2	21.5	37.6	41	35.325	
5	03-11-2006	14:40:19	41.1	21.5	37.8	41.2	35.4	
6	03-11-2006	14:40:20	41.5	21.5	37.9	41.3	35.55	
7	03-11-2006	14:40:21	41.2	21.5	38.1	41.4	35.55	
8	03-11-2006	14:40:22	41.8	21.5	38.2	41.6	35.775	
9	03-11-2006	14:40:23	41.5	21.5	38.4	41.8	35.8	
10	03-11-2006	14:40:24	42.3	21.5	38.4	41.9	36.025	
11	03-11-2006	14:40:25	42.4	21.5	38.6	42.1	36.15	
12	03-11-2006	14:40:26	42.5	21.5	38.8	42.2	36.25	
13	03-11-2006	14:40:27	43	21.5	38.9	42.3	36.425	
14	03-11-2006	14:40:28	43.2	21.5	39	42.5	36.55	
15	03-11-2006	14:40:29	43.3	21.4	39.2	42.7	36.65	
16	03-11-2006	14:40:30	43	21.5	39.3	42.8	36.65	
17	03-11-2006	14:40:31	43.4	21.5	39.5	43	36.85	
18	03-11-2006	14:40:32	43.7	21.5	39.6	43	36.95	
19	03-11-2006	14:40:33	43.7	21.5	39.7	43.2	37.025	
20	03-11-2006	14:40:34	43.9	21.5	39.9	43.3	37.15	
21	03-11-2006	14:40:35	44	21.5	40	43.4	37.225	
22	03-11-2006	14:40:36	44.3	21.5	40.1	43.6	37.375	
23	03-11-2006	14:40:37	44.8	21.4	40.2	43.8	37.55	
24	03-11-2006	14:40:38	44.7	21.5	40.3	43.9	37.6	
25	03-11-2006	14:40:39	44.9	21.5	40.4	44.1	37.725	
26	03-11-2006	14:40:40	44.9	21.4	40.6	44.2	37.775	
27	03-11-2006	14:40:41	45.3	21.5	40.7	44.3	37.95	
28	03-11-2006	14:40:42	45.7	21.5	40.9	44.5	38.15	
29	03-11-2006	14:40:43	45.8	21.4	41.1	44.7	38.25	
30	03-11-2006	14:40:44	46.2	21.4	41.2	44.7	38.375	
31	03-11-2006	14:40:45	46.5	21.4	41.3	44.9	38.525	
32	03-11-2006	14:40:46	46.4	21.5	41.5	45	38.6	
33	03-11-2006	14:40:47	46.5	21.4	41.6	45.2	38.675	
34	03-11-2006	14:40:48	47	21.4	41.7	45.3	38.85	
35	03-11-2006	14:40:49	46.7	21.5	41.9	45.5	38.9	
36	03-11-2006	14:40:50	47.1	21.4	42	45.6	39.025	
37	03-11-2006	14:40:51	47.5	21.5	42.1	45.7	39.2	
38	03-11-2006	14:40:52	47.5	21.4	42.3	45.9	39.275	

Tableau

La propriété AutoFilter permet d'établir un filtre simple, portant sur une ou plusieurs colonnes d'une table de valeurs et pouvant contenir jusqu'à deux conditions par colonne. De la forme :

Function AutoFilter(*[Field As Integer], [Criteria1 As String], [Operator As XlAutoFilterOperator = xlAnd], [Criteria2 As String], [VisibleDropDown As Boolean]*) **As AutoFilter**

Où *Field* est le numéro de la colonne où doit s'appliquer le filtre. Il ne s'agit pas d'une référence absolue mais du numéro d'ordre de la colonne dans la plage de valeur.

Criteria1 et *Criteria2* sont des chaînes de caractères explicitant le(s) critère(s). *Operator* précise si les règles sont composées en ET (xlAnd) ou en OU (xlOr) s'il y a deux critères ; ou prend une valeur particulière de xlBottom10Items, xlBottom10Percent, xlTop10Items, xlTop10Percent.

Si *VisibleDropDown* est vrai, les flèches de filtre sont affichées.

Pour enlever les filtres d'une colonne, on utilise la méthode Autofilter en précisant uniquement l'argument Field, pour enlever tous les filtres, on utilise AutoFilter sans argument.

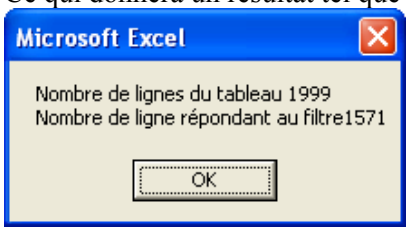
Imaginons dans notre exemple que nous voulions rechercher les zones de la plage où :

30 < T1 < 40 et Moyenne > 67

```
Sub AppliqueFiltre()
    Dim PlageFiltree As Range, PlageBase As Range, cmptLigne As Long
    Dim Zone As Range, Message As String

    With ThisWorkbook.Worksheets("Tableau")
```

```
        Set PlageBase = .Range(.Cells(1, 1), .Cells(1,
1).End(xlDown)).Resize(, 7)
    End With
    With PlageBase
        'enlève les éventuels anciens filtre
        .AutoFilter
        'filtre la colonne T1 (troisième colonne)
        .AutoFilter Field:=3, Criteria1:=">30", Operator:=xlAnd,
Criteria2:="<40", VisibleDropDown:=False
        'filtre la colonne moyenne (septième colonne)
        .AutoFilter Field:=7, Criteria1:=">67"
    End With
    Set PlageFiltree = PlageBase.SpecialCells(xlCellTypeVisible)
    For Each Zone In PlageFiltree.Areas
        cmptLigne = cmptLigne + Zone.Rows.Count
    Next
    Message = "Nombre de lignes du tableau " & PlageBase.Rows.Count &
vbCrLf
    Message = Message & "Nombre de lignes répondant au filtre" & cmptLigne
    MsgBox Message

End Sub
```

Ce qui donnera un résultat tel que :



	A	B	C	D	E	F	G	H
1	Date	Heure	T° 1	T°	T°	T°	oyenne T	4
370	03-11-2006	14:46:24	33.8	82.3	73.8	79.2	67.275	
371	03-11-2006	14:46:25	33	82.5	73.9	79.2	67.15	
372	03-11-2006	14:46:26	36.4	83.2	74	79.3	68.225	
373	03-11-2006	14:46:27	36.7	82.8	74.1	79.3	68.225	
374	03-11-2006	14:46:28	38	82.8	74.1	79.4	68.575	
402	03-11-2006	14:46:56	37.6	84.7	75.5	80.9	69.675	
403	03-11-2006	14:46:57	34.4	85.2	75.6	81	69.05	
404	03-11-2006	14:46:58	30.4	85.2	75.7	81	68.075	
407	03-11-2006	14:47:01	30.5	85.9	75.8	81.2	68.35	
408	03-11-2006	14:47:02	32.3	85.3	75.9	81.2	68.675	
409	03-11-2006	14:47:03	30.7	85.3	76	81.3	68.325	
411	03-11-2006	14:47:05	30.5	85.7	76	81.4	68.4	
419	03-11-2006	14:47:13	32.8	85.8	76.4	81.7	69.175	
420	03-11-2006	14:47:14	30.3	85.6	76.4	81.8	68.525	
425	03-11-2006	14:47:19	31.7	85.6	76.6	82	68.975	
426	03-11-2006	14:47:20	32.6	85.8	76.7	82.1	69.3	
427	03-11-2006	14:47:21	31	85.7	76.7	82.2	68.9	
429	03-11-2006	14:47:23	30.1	85.8	76.8	82.2	68.725	
430	03-11-2006	14:47:24	30.1	85.7	76.8	82.3	68.725	
431	03-11-2006	14:47:25	31.8	85.8	76.9	82.3	69.2	
433	03-11-2006	14:47:27	31.8	85.9	77	82.4	69.275	
434	03-11-2006	14:47:28	30.7	86.1	77	82.5	69.075	
435	03-11-2006	14:47:29	31	86	77.1	82.5	69.15	
436	03-11-2006	14:47:30	31	85.7	77.2	82.5	69.1	
437	03-11-2006	14:47:31	30.6	86	77.2	82.6	69.1	
438	03-11-2006	14:47:32	33.8	86.2	77.2	82.7	69.975	
439	03-11-2006	14:47:33	34	86.2	77.3	82.7	70.05	
441	03-11-2006	14:47:35	31.4	86	77.3	82.7	69.35	
442	03-11-2006	14:47:36	31.8	86.1	77.4	82.8	69.525	
443	03-11-2006	14:47:37	31.9	86.1	77.4	82.8	69.55	
444	03-11-2006	14:47:38	30.3	85.8	77.5	82.8	69.1	
445	03-11-2006	14:47:39	31.7	85.5	77.5	82.9	69.4	
446	03-11-2006	14:47:40	31.3	86	77.5	82.9	69.425	
447	03-11-2006	14:47:41	30.3	86.3	77.5	83	69.275	
448	03-11-2006	14:47:42	33	86.2	77.6	83	69.95	
449	03-11-2006	14:47:43	31.7	86.4	77.6	83.1	69.7	
450	03-11-2006	14:47:44	32.1	86.1	77.6	83.1	69.725	

Il existe aussi une méthode AdvancedFilter encore plus sophistiquée mais qui demande de gérer les plages de critères, ce qui nous emmènerait un peu loin.

AutoFill, FillDown, FillUp, FillLeft & FillRight

Les propriétés Fill[Direction] recopient le contenu **et le format** de la cellule de base dans la direction spécifiée. La cellule de base se définit comme :

La cellule la plus haute pour FillDown
La cellule la plus basse pour FillUp
La cellule la plus à gauche pour FillRight
La cellule la plus à droite pour FillLeft

La propriété Autofill effectue une recopie incrémentée si c'est possible, une recopie standard si non. De la forme :

Function AutoFill(*Destination As Range, [Type As XlAutoFillType = xlFillDefault]*) **As Variant**

Destination est la plage où doit être effectuée la recopie, elle doit obligatoirement contenir la ou les cellules sources (celles qui contiennent les valeurs initiales). *Type* indique le type de recopie, il peut prendre les valeurs :

xlFillDays	Incrémente les jours si la cellule source est une date. Sinon exécute une recopie standard
xlFillFormats	Recopie les formats mais pas le contenu
xlFillSeries	Crée une série avec l'incrément standard ou l'incrément déduit
xlFillWeekdays	Incrémente sur les jours ouvrables
xlGrowthTrend	Incrément sur un incrément déduit multiplicatif
xlFillCopy	Recopie standard. Utilisez plutôt une méthode Fill
xlFillDefault	Laisse Excel déterminer le type de recopie
xlFillMonths	Incrémente les mois si la cellule source est une date. Sinon exécute une recopie standard
xlFillValues	Incrémente selon l'incrément déduit sans recopier les formules
xlFillYears	Incrémente les années si la cellule source est une date. Sinon exécute une recopie standard
xlLinearTrend	Incrémente selon l'incrément déduit ou l'incrément standard

Pour bien comprendre comment utiliser cette méthode, exposons d'abord ces règles de fonctionnement. Pour faire une recopie 'incrémentée' il faut à minima définir, une valeur de départ, une règle d'incrémentation et un incrément. La valeur de départ va donc être l'objet Range appelant la méthode. La règle d'incrémentation sera définie par l'argument *Type,* il reste à définir l'incrément. Excel peut utiliser dans la plupart des cas l'incrément par défaut qui vaut 1, pour peu qu'il n'y ait pas d'ambiguïté. Dès lors que celui-ci ne convient pas, il va falloir lui donner un autre incrément. On utilise pour cela ce qu'on appelle un incrément déduit en passant à Excel deux valeurs dans la plage source, la valeur de départ, et la première valeur incrémentée. Regardons le code suivant :

```
Sub TestAutofill()

Dim MaPlage As Range

    Set MaPlage = ThisWorkbook.Worksheets("Methode").Cells(1, 1)
    'cas 1
    MaPlage.Value = CDate("28/01/2002")
    MaPlage.Offset(1).Value = CDate("28/02/2002")
    MaPlage.Resize(2).AutoFill MaPlage.Resize(30), xlFillDefault
    'cas 2
    MaPlage.Offset(, 1).Value = CDate("28/01/2002")
    MaPlage.Offset(, 1).AutoFill MaPlage.Offset(, 1).Resize(30),
xlFillMonths
    'cas 3
    With ThisWorkbook.Worksheets("Methode")
```

```
        .Range("C1").Value = 1
        .Range("C1").AutoFill .Range("C1:C30"), xlFillSeries
    End With
    'cas 4
    With ThisWorkbook.Worksheets("Methode")
        .Range("D1").Value = 1
        .Range("D2").Value = 3
        .Range("D1:D2").AutoFill .Range("D1:D30"), xlFillSeries
    End With
    'cas 5
    With ThisWorkbook.Worksheets("Methode")
        .Range("E1").Value = 1
        .Range("E2").Value = 3
        .Range("E1:E2").AutoFill .Range("E1:E30"), xlLinearTrend
    End With
    'cas 6
    With ThisWorkbook.Worksheets("Methode")
        .Range("F1").Value = 1
        .Range("F2").Value = 3
        .Range("F1:F2").AutoFill .Range("F1:F30"), xlGrowthTrend
    End With

End Sub
```

Ce Code donnera le résultat suivant :

	A	B	C	D	E	F	G
1	28/01/2002	28/01/2002	1	1	1	1	
2	28/02/2002	28/02/2002	2	3	3	3	
3	28/03/2002	28/03/2002	3	5	5	9	
4	28/04/2002	28/04/2002	4	7	7	27	
5	28/05/2002	28/05/2002	5	9	9	81	
6	28/06/2002	28/06/2002	6	11	11	243	
7	28/07/2002	28/07/2002	7	13	13	729	
8	28/08/2002	28/08/2002	8	15	15	2187	
9	28/09/2002	28/09/2002	9	17	17	6561	
10	28/10/2002	28/10/2002	10	19	19	19683	
11	28/11/2002	28/11/2002	11	21	21	59049	
12	28/12/2002	28/12/2002	12	23	23	177147	
13	28/01/2003	28/01/2003	13	25	25	531441	
14	28/02/2003	28/02/2003	14	27	27	1594323	
15	28/03/2003	28/03/2003	15	29	29	4782969	
16	28/04/2003	28/04/2003	16	31	31	14348907	
17	28/05/2003	28/05/2003	17	33	33	43046721	
18	28/06/2003	28/06/2003	18	35	35	129140163	
19	28/07/2003	28/07/2003	19	37	37	387420489	
20	28/08/2003	28/08/2003	20	39	39	1162261467	
21	28/09/2003	28/09/2003	21	41	41	3486784401	
22	28/10/2003	28/10/2003	22	43	43	1.046E+10	
23	28/11/2003	28/11/2003	23	45	45	3.1381E+10	
24	28/12/2003	28/12/2003	24	47	47	9.4143E+10	
25	28/01/2004	28/01/2004	25	49	49	2.8243E+11	
26	28/02/2004	28/02/2004	26	51	51	8.4729E+11	
27	28/03/2004	28/03/2004	27	53	53	2.5419E+12	
28	28/04/2004	28/04/2004	28	55	55	7.6256E+12	
29	28/05/2004	28/05/2004	29	57	57	2.2877E+13	
30	28/06/2004	28/06/2004	30	59	59	6.863E+13	
31							
32							
33							

Dans le cas 1, nous laissons Excel déterminer le type d'incrémentation. Dés lors, il y a obligation de fournir deux valeurs pour que celle-ci réussisse. Comme il y a un mois d'écart entre les deux dates, Excel incrémente par pas d'un mois. Nous obtenons le même résultat avec le cas deux, mais comme l'incrément par défaut est de un et que nous précisons une incrémentation en mois, une seule valeur suffit.

Dans le cas 3, nous demandons une incrémentation de type série avec un incrément standard. Une seule valeur suffit donc. Dans le cas 4, nous voulons le même type d'incrémentation mais avec un incrément égal à 2. Nous sommes donc obligés de passer deux valeurs pour définir l'incrément. L'incrémentation de type série étant linéaire, nous avons le même résultat dans le cas 5. Par contre l'incrémentation en croissance interprète l'incrément différemment. Dans une incrémentation de ce type, Excel cherche un pas multiplicatif. Comme il faut multiplier par 3 la première valeur pour obtenir la seconde, Excel multipliera par trois chaque valeur pour obtenir la valeur suivante.

AutoFit

Mets la ou les lignes / colonnes en mode d'ajustement automatique. La plage appelante doit forcément contenir des lignes ou des colonnes entières. Pour que l'ajustement se déroule correctement, il faut que les cellules contiennent les valeurs avant l'appel de la méthode

BorderAround

Gère l'encadrement extérieur de la plage. De la forme :

Function BorderAround(*[LineStyle], [Weight As XlBorderWeight = xlThin], [ColorIndex As XlColorIndex = xlColorIndexAutomatic], [Color As Long]*)

Par exemple, pour encadrer d'un trait pointillé épais en rouge la plage "B2:E6", nous pouvons écrire :

```
ThisWorkbook.Worksheets("Methode").Range("B2").Resize(5, 4).BorderAround
xlDash, xlMedium, Color:=vbRed
```

Calculate

Déclenche le calcul pour la plage appelante. Attention, les antécédents n'étant pas calculés par défaut, le résultat peut être erroné.

Clear, ClearComments, ClearContents & ClearFormats

Méthodes d'effacements. La méthode Clear efface tout, les méthodes spécifiques effacent l'élément spécifique.

ColumnDifferences & RowDifferences

Ces méthodes peuvent être assez intéressantes même si elles sont un peu complexes à comprendre. En fait, ces méthodes servent à récupérer les cellules ayant une valeur différente dans la dimension spécifiée, sachant que sont exclues d'office les cellules n'ayant pas de valeur. De la forme :

Function ColumnDifferences(*Comparison As Range*) **As Range**

Function RowDifferences(*Comparison As Range*) **As Range**

L'argument *Comparison* est un objet Range contenant une seule cellule appartenant à la plage de recherche. Idéalement la plage de recherche ne contient qu'une colonne pour ColumnDifferences ou qu'une ligne pour RowDifferences. Si la plage contient plusieurs colonnes, Excel va gérer le cas comme si vous appeliez plusieurs fois consécutivement la méthode pour chaque colonne en décalant implicitement la cellule de comparaison. Évitez donc de faire cela et travaillez sur des plages d'une colonne (ou ligne).

Donc prenons l'exemple suivant en repartant de la feuille tableau que nous avons utilisé pour les exemples d'Autofilter.

Je souhaite connaître toutes les cellules de la plage "D2:D1000" qui ont une valeur différente de celle de la plage "D2".

```
Sub TestMethod()
Dim MaPlage As Range
    Set MaPlage = ThisWorkbook.Worksheets("Tableau").Range("D2")
    Set MaPlage = MaPlage.Resize(999).ColumnDifferences(MaPlage)
    MaPlage.Select
End Sub
```

Comme vous voyez, c'est assez simple. Nous verrons dans la discussion technique à quoi cela peut servir.

Cut & Copy

Gère le Copier-Coller ou le Couper-Coller d'une plage de cellule. De la forme :

Function Copy(*[Destination As Range]*)

Function Cut(*[Destination As Range]*)

Où *Destination* indique la plage de destination. Celle-ci doit contenir ou une cellule unique qui sera le coin supérieur gauche de la plage de collage, ou une plage de même dimension que la plage appelante. Si tel n'est pas le cas, la cellule supérieure gauche de la plage de destination sera utilisée sans tenir compte de la plage éventuellement passée. Si *Destination* est omis, la plage copiée est placée dans le presse-papier. La destination peut être dans une autre feuille ou dans un autre classeur. Il peut y avoir un message d'alerte si les cellules de destination contiennent déjà des valeurs.

```
Sub TestCopy()

Dim MaPlage As Range

    Set MaPlage = ThisWorkbook.Worksheets("Tableau").Range("A1:A50")
    MaPlage.Copy ThisWorkbook.Worksheets("Depend").Range("P10")

End Sub
```

DataSeries

Cette méthode est équivalente à la méthode Autofill si ce n'est que vous pouvez précisez la valeur du pas et la dernière valeur et donc vous affranchir de la saisie de plusieurs valeurs. De la forme :

Function DataSeries([Rowcol], [Type As XlDataSeriesType = xlDataSeriesLinear], [Date As XlDataSeriesDate = xlDay], [Step], [Stop], [Trend])

L'argument RowCol peut prendre les valeurs xlRows ou xlColumns selon la direction que vous souhaitez pour la plage. Si cet argument est omis, Excel interprètera la plage appelante pour utiliser la dimension la plus grande.

Le code suivant créera une série incrémentée par pas de 2 sur la plage D1:D50.

```
Dim MaPlage As Range

    Set MaPlage = ThisWorkbook.Worksheets("Methode").Range("D1:F50")
    MaPlage.Cells(1).Value = 1
    MaPlage.DataSeries , xlDataSeriesLinear, , 2
```

Delete

Supprime les cellules de la plage. La méthode attends un argument Shift pouvant prendre les valeurs **xlShiftToLeft** ou **xlShiftUp** selon la direction du décalage souhaité (dans une suppression les cellules supprimées sont toujours remplacées soit par les cellules situées au dessous, soit par les cellules situées à droite). Pour supprimer des lignes entières ou des colonnes entières, utilisez EntireRow ou EntireColumn.

Find, FindNext & FindPrevious

Méthode de recherche portant sur des cellules contenues dans la plage appelante. De la forme :

Function Find(*What As Variant, [After As Range], [LookIn As XlFindLookIn], [LookAt As XlLookAt], [SearchOrder As XlSearchOrder], [SearchDirection As XlSearchDirection = xlNext], [MatchCase As Boolean], [MatchByte], [SearchFormat As Boolean]*) **As Range**

Où *What* est la valeur cherchée. L'argument *After* précise la cellule de début de recherche. Attention, la recherche commence **après** cette cellule et non par elle.

L'argument *LookIn* précise si la recherche à lieu dans les valeurs de la plage (xlValues), dans les formules (xlFormulas) ou dans les commentaires (xlComments).

L'argument *LookAt* précise si la recherche compare l'élément à la totalité du contenu (xlWhole) ou à une partie du contenu (xlPart).

L'argument *SearchOrder* précise la direction de recherche choisie, par colonne (xlByColumns) ou par ligne (xlByRows).

L'argument *SearchDirection* précise le sens de la recherche, vers l'avant (xlNext) c'est-à-dire vers le bas ou vers la droite selon la direction choisie, vers l'arrière (xlPrevious) c'est-à-dire vers le haut ou vers la gauche.

MatchCase précise si la recherche tient compte de la casse (majuscule / minuscule) ou non.

Les deux autres arguments ne seront pas utilisés dans ce cours.

Les méthodes FindNext et FindPrevious n'attendent que l'argument After puisqu'elles utilisent implicitement l'ensemble des arguments précisés dans l'appel de la méthode Find.

Le principe de la recherche est assez simple. La plage appelante sera la plage de recherche. Si la valeur existe au moins une fois, la fonction Find renverra un objet Range correspondant à la première cellule trouvée, sinon elle renverra Nothing. FindNext et FindPrevious pourront alors être appelés pour rechercher si d'autres cellules contiennent la valeur. La recherche est circulaire dans la plage. Vous devez donc stocker l'adresse de la première cellule correspondante le cas échéant pour bloquer le processus. Par exemple, le code suivant va rechercher l'ensemble des cellules contenant la valeur 82 dans la plage appelante et mettre le fond de la cellule en bleu.

```
Sub Find82()

Dim MaPlage As Range, Trouve As Range, Adresse1 As String

    With ThisWorkbook.Worksheets("Tableau")
        Set MaPlage = .Range(.Cells(1, 3), .Cells(.Rows.Count,
6).End(xlUp))
    End With
    Set Trouve = MaPlage.Find(82, MaPlage.Cells(1), xlValues, xlWhole,
xlByRows, xlNext)
    If Not Trouve Is Nothing Then
        Adresse1 = Trouve.Address
        Do
            Trouve.Interior.Color = vbBlue
            Set Trouve = MaPlage.FindNext(Trouve)
        Loop While StrComp(Adresse1, Trouve.Address) <> 0
    End If

End Sub
```

Insert

C'est la méthode inverse de Delete. Elle attend le même argument *Shift* pour préciser le sens du décalage, vers le bas ou à droite, la remarque pour les lignes ou colonnes entières restant valable.

Merge & UnMerge

Gère la fusion des cellules de la plage. La méthode accepte un argument booléen Across que l'on peut utiliser lorsque la plage contient plusieurs lignes et plusieurs colonnes. Si Across est vrai, les cellules sont fusionnées par ligne, sinon toute la plage est fusionnée en une seule cellule. Si les cellules contiennent des valeurs, vous aurez l'apparition d'un message d'alerte prévenant que seule la valeur la plus à gauche sera conservée. Vous pouvez le désactiver, comme dans :

```
    Set MaPlage = ThisWorkbook.Worksheets("methode").Range("G2:J10")
    Application.DisplayAlerts = False
    MaPlage.Merge True
    Application.DisplayAlerts = True
```

La méthode UnMerge sépare les cellules fusionnées, elle n'accepte pas d'argument.

PasteSpecial

La méthode collage spécial reste très intéressante pour un certain nombre d'opérations. Elle fonctionne forcément en conjonction avec la méthode Copy. Si vous avez bien suivi ce que nous avons vu précédemment, elle n'est pas très utile pour le collage spécial de type valeur ou de type formule puisqu'il est plus simple de passer par les propriétés tableau telles que Value ou Formula.

De la forme :

Function PasteSpecial(*[Paste As XlPasteType = xlPasteAll], [Operation As XlPasteSpecialOperation = xlPasteSpecialOperationNone], [SkipBlanks As Boolean], [Transpose As Boolean]*)

Le paramètre *Paste* définit le type de collage, il peut prendre les valeurs :

xlPasteAll
xlPasteAllExceptBorders
xlPasteColumnWidths
xlPasteComments
xlPasteFormats
xlPasteFormulas
xlPasteFormulasAndNumberFormats
xlPasteValidation
xlPasteValues
xlPasteValuesAndNumberFormats

L'argument *Operation* définit les opérations spécifiques. Il peut prendre les valeurs :

xlPasteSpecialOperationAdd
xlPasteSpecialOperationDivide
xlPasteSpecialOperationMultiply
xlPasteSpecialOperationNone
xlPasteSpecialOperationSubtract

Ces opérations de collage n'ont de sens que pour les collages de valeur, lorsque la plage de destination contient aussi des valeurs. Si l'argument vaut xlPasteSpecialOperationNone, les valeurs existantes sont écrasées par les valeurs collées, sinon Excel exécute l'opération précisée entre la valeur existante et la valeur collée.

L'argument SkipBlank est un booléen qui précise si les cellules vides doivent être ignorées lors du collage.

L'argument Transpose est un booléen qui définit si la plage doit être transposée, autrement dit si les deux dimensions ligne / colonne doivent être inversée.

On utilise donc cette méthode, soit pour coller des formats, soit pour coller des valeurs et des formats, soit enfin pour transposer les plages.

On n'utilise finalement qu'assez peu cette méthode pour réaliser des opérations car les scénatii mettant en œuvre cette fonctionnalité sont rares.

Replace

La méthode Replace fonctionne à peu près comme la méthode Find, si ce n'est qu'on précise un argument Replacement qui contient la valeur de remplacement.

Sort

Cette méthode gère le tri dans les plages. Elle est assez souvent mal comprise et donc mal utilisée, aussi allons nous la détailler un peu avant de voir son écriture.

Le tri dans une plage rectangulaire a une orientation, soit on tri par colonne c'est-à-dire que les cellules restent dans une colonne fixe mais peuvent changer de ligne, soit on tri en ligne et les lignes restent fixes alors que l'ordre dans les colonnes peut être modifié.

Le tri Excel accepte jusqu'à trois clés de tri. Une clé de tri, définit une ligne ou une colonne qui va subir le tri, entraînant de ce fait le tri des autres éléments de la même dimension. Prenons notre exemple de tableau désormais classique pour mieux visualiser le principe.

	A	B	C	D	E	F	G
1	Date	Heure	T° 1	T° 2	T° 3	T° 4	Moyenne T1-T4
2	03/11/2006	14:40:16	27.1	21.5	37.4	40.7	31.675
3	03/11/2006	14:40:17	27.8	21.5	37.5	40.8	31.9
4	03/11/2006	14:40:18	28	21.5	37.6	41	32.025
5	03/11/2006	14:40:19	28.1	21.5	37.8	41.2	32.15
6	03/11/2006	14:40:20	28.3	21.5	37.9	41.3	32.25
7	03/11/2006	14:40:21	28.4	21.5	38.1	41.4	32.35
8	03/11/2006	14:40:22	28.5	21.5	38.2	41.6	32.45
9	03/11/2006	14:40:23	28.5	21.5	38.4	41.8	32.55
10	03/11/2006	14:40:24	28.7	21.5	38.4	41.9	32.625
11	03/11/2006	14:40:25	28.7	21.5	38.6	42.1	32.725
12	03/11/2006	14:40:26	28.8	21.5	38.8	42.2	32.825
13	03/11/2006	14:40:27	28.8	21.5	38.9	42.3	32.875
14	03/11/2006	14:40:28	28.8	21.5	39	42.5	32.95
15	03/11/2006	14:40:29	28.9	21.4	39.2	42.7	33.05
16	03/11/2006	14:40:30	29	21.5	39.3	42.8	33.15
17	03/11/2006	14:40:31	29.1	21.5	39.5	43	33.275
18	03/11/2006	14:40:32	29.1	21.5	39.6	43	33.3
19	03/11/2006	14:40:33	29.2	21.5	39.7	43.2	33.4
20	03/11/2006	14:40:34	29.2	21.5	39.9	43.3	33.475
21	03/11/2006	14:40:35	29.2	21.5	40	43.4	33.525
22	03/11/2006	14:40:36	29.3	21.5	40.1	43.6	33.625
23	03/11/2006	14:40:37	29.3	21.4	40.2	43.8	33.675
24	03/11/2006	14:40:38	29.5	21.5	40.3	43.9	33.8
25	03/11/2006	14:40:39	29.5	21.5	40.4	44.1	33.875
26	03/11/2006	14:40:40	29.5	21.4	40.6	44.2	33.925
27	03/11/2006	14:40:41	29.5	21.5	40.7	44.3	34
28	03/11/2006	14:40:42	29.5	21.5	40.9	44.5	34.1
29	03/11/2006	14:40:43	29.5	21.4	41.1	44.7	34.175
30	03/11/2006	14:40:44	29.6	21.4	41.2	44.7	34.225
31	03/11/2006	14:40:45	29.6	21.4	41.3	44.9	34.3

Les données du tableau étant ordonnées en colonne, nous allons forcément trier dans ce sens. Si nous décidons de trier uniquement T°1, l'ordre des cellules de la colonne C va changer, mais l'ordre des autres colonnes ne va pas bouger. Ceci ne serait pas tellement logique puisque chaque ligne à une cohérence propre. Nous devons donc trier tout le tableau. Lorsque nous allons trier T°1 (la colonne C) l'ordre de toutes les lignes va donc se modifier pour suivre le nouvel ordre des valeurs de T°1. Cette colonne sera donc la clé de tri d'un objet Range représentant tout le tableau.

Je vous ai dit qu'il pouvait y avoir jusqu'à trois clés, que se passe-t-il donc lorsqu'on précise une seconde clé, par exemple T°2 ?

Les lignes vont d'abord se réordonner selon le tri de la première clé, par exemple par valeurs de T°1 croissante. Comme il n'est pas possible de trier parfaitement la deuxième clé sans perdre l'ordre défini par la première, Excel va procéder à un tri secondaire, c'est-à-dire que les lignes ne seront réordonnées selon la valeur de T°2 qu'à valeur de T°1 équivalente. Si nous ajoutons une troisième clé, le tri ne portera que sur les lignes ayant des valeurs de T°1 et de T°2 équivalente.

Lorsque vous allez demander un tri à Excel, vous allez aussi devoir lui signaler si la plage à une colonne / ligne de titre ou non, puisque celle-ci ne doit évidemment pas être triée. Regardons maintenant la syntaxe de la méthode. En écriture exhaustive, celle-ci est de la forme :

Function Sort(*[Key1], [Order1 As XlSortOrder = xlAscending], [Key2], [Type], [Order2 As XlSortOrder = xlAscending], [Key3], [Order3 As XlSortOrder = xlAscending], [Header As XlYesNoGuess = xlNo], [OrderCustom], [MatchCase], [Orientation As XlSortOrientation = xlSortRows], [SortMethod As XlSortMethod = xlPinYin], [DataOption1 As XlSortDataOption = xlSortNormal], [DataOption2 As XlSortDataOption = xlSortNormal], [DataOption3 As XlSortDataOption = xlSortNormal]*)

Un certain nombre de ces arguments ne servent que pour le tri des tableaux croisés où pour des tris spéciaux que nous ne verrons pas dans ce cours, je vais donc écrire la définition de la méthode sous la forme :

Function Sort(*[Key1], [Order1 As XlSortOrder = xlAscending], [Key2], [Order2 As XlSortOrder = xlAscending], [Key3], [Order3 As XlSortOrder = xlAscending], [Header As XlYesNoGuess = xlNo], [MatchCase], [Orientation As XlSortOrientation = xlSortRows]*)

Ecrite ainsi, nous allons retrouver tout ce que nous avons vu précédemment.

Les arguments contiennent d'abord un jeu de trois clés, sous la forme d'un argument *Key* qui contient la première cellule de la clé contenant une valeur à trier (pas un titre) puis un argument *Order* qui définit le sens du tri (croissant ou décroissant).

Nous trouvons ensuite un argument *Header* qui précise s'il y a une zone de titre ou non. Croyez en ma vieille expérience, prenez l'habitude de sélectionner des zones à trier sans ligne de titre, vous vous simplifierez l'existence.

L'argument *MatchCase* est un booléen qui précise si le tri doit respecter ou non la casse lorsqu'on tri des chaînes de caractères.

Enfin l'argument *Orientation* détermine le sens du tri. Normalement il accepte deux valeurs (XlColumns ou XlRows) mais pour des raisons de compréhension, on utilise parfois les constantes équivalentes xlTopToBottom ou xlLeftToRight.

Si vous n'êtes pas parti vous lobotomiser à coup de tisonnier, vous allez voir qu'à utiliser maintenant c'est simple comme chou. Nous voulons trier notre tableau selon la valeur de la moyenne, puis selon l'ordre de T°1.

```
Sub TestTri()
    Dim maplage As Range
    'définit une plage contenant tout le tableau
    With ThisWorkbook.Worksheets("Tri")
        Set maplage = .Range(.Cells(1, 1), .Cells(.Rows.Count,
7).End(xlUp))
    End With
    'redéfinit une plage sans ligne de titre
    Set maplage = maplage.Offset(1).Resize(maplage.Rows.Count - 1)
    'voir explication suivant l'exemple
    maplage.Sort key1:=maplage.Cells(6), order1:=xlAscending,
key2:=maplage.Cells(3), order2:=xlAscending, header:=xlNo,
Orientation:=xlTopToBottom
End Sub
```

La première chose est donc se générer un objet Range contenant l'ensemble du tableau d'une manière tout à fait classique. Pour simplifier le tri, je vais décaler l'objet Range pour enlever la ligne de titre. Pour cela il suffit de décaler d'une ligne vers le bas la plage est de diminuer sa taille d'une ligne. Ensuite de quoi nous allons trier. Nous avons donc besoin de définir deux clés de tri, la première sur la moyenne et la seconde sur la colonne T°1. Comme je vous l'ai dit plus haut, la définition d'une clé de tri consiste à passer une paire d'argument dont le premier est la première cellule contenant une valeur n'étant pas un titre de la colonne et le second étant le sens du tri. Comme nous avons enlevé physiquement la ligne de titre, la première ligne de la plage contient des valeurs. Comme nous l'avons vu aussi précédemment, je peux utiliser la propriété Cells de l'objet Range pour renvoyer une cellule particulière en utilisant une coordonnée linéaire. Les cellules de la première ligne vont avoir une coordonnée comprise entre 1 et Columns.Count. Dans notre cas, les deux colonnes qui nous intéressent sont la sixième de la plage, et la troisième. Donc les deux clés de tri sont la sixième et la troisième de la plage. L'argument Header sera à xlNo puisque nous avons enlevé les titres, et Orientation vaudra xlColumns ou son équivalent xlTopToBottom. Ce n'est pas plus compliqué que cela.

SpecialCells

Attention, ce passage de l'aide en ligne est un sommet de la traduction raté. Les explications sont ou fausses ou sans aucun sens. Nous allons voir ici le fonctionnement réel de cette méthode, ne tenez pas compte de l'aide en ligne.

Cette méthode permet de renvoyer une sous plage de cellules particulières selon les arguments passés. De la forme :

Function SpecialCells(*Type As XlCellType, [Value As XlSpecialCellsValue]*) **As Range**

L'argument *Type* peut prendre une des valeurs suivantes :

Constantes	Explication
xlCellTypeAllFormatConditions	Renvoie toutes les cellules de la feuille ayant les mêmes règles de format conditionnel que la cellule supérieure gauche de la plage appelante. Celle-ci **doit** avoir un format conditionnel.
xlCellTypeAllValidation.	Renvoie toute les cellules de la plage appelante ayant une validation définie.
xlCellTypeBlanks	Renvoie toutes les cellules vides de la plage appelante.
xlCellTypeComments	Renvoie toutes les cellules de la plage appelante contenant des commentaires.
xlCellTypeConstants	Renvoie toutes les cellules de la plage appelante contenant des valeurs mais pas de formules
xlCellTypeFormulas	Renvoie toutes les cellules de la plage appelante contenant des formules
xlCellTypeLastCell	Renvoie la dernière cellule de la feuille quelle que soit la plage appelante. La dernière cellule ne contient pas forcément une valeur, elle est définie comme étant l'intersection de la dernière ligne utilisée et de la dernière colonne utilisée
xlCellTypeSameFormatConditions	Renvoie toutes les cellules de la plage appelante ayant un format conditionnel.
xlCellTypeSameValidation	Si la plage appelante ne contient qu'une cellule ayant une validation définie, renvoie toutes les cellules de la feuille ayant les mêmes critères de validation. Sinon renvoie toutes les cellules de la plage appelante ayant les mêmes critères de validation que la cellule supérieure gauche de la plage appelante.
xlCellTypeVisible	Renvoie toutes les cellules visibles de la plage appelante. Les cellules visibles sont définies comme toutes cellules dont la ligne ou la colonne n'est pas masquée.

Lorsque le type à pour valeur **xlCellTypeConstants** ou **xlCellTypeFormulas**, vous pouvez passer un argument optionnel Value qui peut prendre une composition (masque binaire) des valeurs suivantes :

xlErrors ⇨ Renvoie uniquement les cellules de la sous plage affichant une valeur d'erreur.
xlLogical ⇨ Renvoie les cellules de la sous plage contenant une valeur logique (Vrai ou Faux)
xlNumbers ⇨ Renvoie les cellules de la sous plage affichant une valeur numérique
xlTextValues ⇨ Renvoie les cellules de la sous plage affichant une chaîne de caractère

Chaque fois qu'il n'existe pas de cellules correspondantes aux critères spécifiés, Excel lèvera une erreur récupérable 1004 ayant pour message « Pas de cellules correspondantes ».

Le code suivant illustre le fonctionnement de ces combinaisons, puis efface les données numériques du tableau sans effacer les formules.

```
Sub CleanSpecial()

    Dim maplage As Range, Feuille As Worksheet

    Set Feuille = ThisWorkbook.Worksheets("Special")
    Set maplage = Feuille.Range("A1:G100")
    Debug.Print maplage.SpecialCells(xlCellTypeFormulas,
xlNumbers).Address(False, False)
    'G2:G100
    On Error Resume Next
    Debug.Print maplage.SpecialCells(xlCellTypeFormulas,
xlTextValues).Address(False, False)
    'lève une erreur
    If Err.Number = 1004 And InStr(1, Err.Description, "correspondante",
vbTextCompare) > 0 Then Err.Clear
    Debug.Print maplage.SpecialCells(xlCellTypeConstants,
xlNumbers).Address(False, False)
    'A2:F100
    Debug.Print maplage.SpecialCells(xlCellTypeConstants,
xlTextValues).Address(False, False)
    'A1:G1
    maplage.SpecialCells(xlCellTypeConstants, xlNumbers).ClearContents

End Sub
```

Discussion technique

Maintenant que nous avons vu une bonne partie du modèle objet de manipulation des feuilles de calcul, nous allons voir ensemble quelques problématiques courantes de la programmation Excel pour se familiariser avec cette programmation et pour voir les diverses solutions qu'on peut envisager face à un problème posé.

Comprendre Excel

In fine, la plus grande difficulté de la programmation Excel reste une connaissance correcte du fonctionnement d'Excel. Cela peut sembler stupide et c'est pourtant la principale cause des difficultés de conception que l'on rencontre. Pour illustrer ce propos, nous allons commencer par un classique de la programmation Excel, la récupération d'un fichier avec suppression de ligne.

Nous allons donc écrire un code qui ouvre un fichier texte définit à l'exécution, qui va supprimer 4 lignes sur cinq puis qui va intégrer les lignes restantes dans la feuille 'données' de notre classeur macro.

Commençons par une approche classique.

```
Public Sub TraiteFichier()

Dim NomFichier As String, ClasseurSource As Workbook, Ligne As Range,
compteur As Long

    NomFichier = Application.GetOpenFilename("Fichier Morphee (*.0*),
*.0*", , "Fichier d'acquisition")
    If StrComp(NomFichier, "Faux", vbTextCompare) = 0 Then Exit Sub
    Application.Workbooks.OpenText Filename:=NomFichier, StartRow:=2,
DataType:=xlDelimited, Tab:=True, DecimalSeparator:="."
    Set ClasseurSource = Application.ActiveWorkbook
    For Each Ligne In ClasseurSource.Worksheets(1).UsedRange.Rows
        compteur = compteur + 1
        If compteur > 1 Then
            Ligne.Delete
        End If
        If compteur = 5 Then compteur = 0
    Next Ligne

End Sub
```

Dans ce code, nous avons donc proposé à l'utilisateur une boîte de sélection de fichiers, mis le classeur ouvert dans une variable objet de type Workbook, puis écrit un traitement de suppression de lignes. La première partie du code ne pose pas de problème, par contre le code de suppression de ligne va en poser.

Il est pourtant juste dans son concept, puisqu'il va parcourir la collection des lignes et en supprimer quatre sur cinq. Pourtant il est fonctionnellement faux puisqu'il contient une faute de programmation et une approche erronée du fonctionnement d'Excel.

La faute de programmation n'est pas évidente. On ne doit pas utiliser la méthode Delete sur le membre d'une collection qu'on est en train d'énumérer. L'énumération est un parcours des membres d'une collection. Si on supprime certains de ces membres pendant le parcours, il n'existe pas de garantie que le parcours se déroulera correctement. En l'occurrence, s'il n'y avait que cette erreur, le code pourrait fonctionner. Mais la deuxième est beaucoup plus grave puisqu'elle ne prend pas en compte le fonctionnement d'Excel.

Que se passe-t-il lorsqu'on supprime une ou plusieurs lignes dans Excel ? Au nom de la continuité de numérotation, les lignes sont remplacées par les lignes situées au dessous, et Excel recrée des lignes en fin de feuille pour que le nombre de lignes de la feuille reste constant.

Or dans notre code, lors de l'énumération, nous allons parcourir les lignes par numéro d'index, c'est-à-dire par numéro de ligne. Lorsque nous allons par exemple supprimer la ligne 2, la ligne 3 va se retrouver en position 2 et la ligne 4 en position 2. Lors de la boucle suivante de l'énumération, la ligne 3 va être supprimée, mais la ligne 2 va être conservée alors qu'elle aurait du être supprimée aussi. Ce code va donc supprimer une ligne sur 2 quatre fois de suite, puis garder deux lignes et ainsi de suite.

Tout ceci pour dire que l'approche choisie est particulièrement mauvaise. Reprenons donc notre conception, pour générer une méthode applicable à tous les cas de suppression, c'est-à-dire x lignes sur n.

Déjà nous ne devons pas énumérer. Nous utiliserons donc une boucle For…Next classique. Si nous ne voulons pas avoir à prendre en compte le glissement des lignes engendré par la suppression, nous devons parcourir la collection dans l'autre sens. Nous aurons donc un pas négatif. Mais dans ce cas, nous devons envisager le cas probable où le nombre de ligne du fichier n'est pas un multiple de n.

Je vais écrire ce code et nous le discuterons par la suite.

```
Public Sub TraiteFichier()

Dim NomFichier As String, ClasseurSource As Workbook, Lignes As Range,
compteur As Long

    NomFichier = Application.GetOpenFilename("Fichier Morphee (*.0*),
*.0*", , "Fichier d'acquisition")
    If StrComp(NomFichier, "Faux", vbTextCompare) = 0 Then Exit Sub
    Application.Workbooks.OpenText Filename:=NomFichier, StartRow:=2,
DataType:=xlDelimited, Tab:=True, DecimalSeparator:="."
    Set ClasseurSource = Application.ActiveWorkbook
    With ClasseurSource.Worksheets(1).UsedRange
        Set Lignes = .Offset(1).Resize(.Rows.Count - 1)
    End With
    Call DeleteXSurN(4, 5, Lignes)
    ClasseurSource.Worksheets(1).UsedRange.Copy
Destination:=ThisWorkbook.Worksheets("Donnee").Cells(1, 1)
    ClasseurSource.Close False

End Sub

Private Sub DeleteXSurN(ByVal X As Integer, ByVal N As Integer, ByRef Plage
As Range)

    Dim NbLigne As Long, LimSup As Long, compteur As Long

    Application.ScreenUpdating = False
    NbLigne = Plage.Rows.Count
    LimSup = NbLigne \ N
    If NbLigne = LimSup * N Then LimSup = LimSup - 1
    For compteur = LimSup * N + 1 To 1 Step -1 * N
        Plage.Rows(compteur).Offset(N - X).Resize(X).Delete
        Application.StatusBar = "lignes " & compteur
    Next compteur
    Application.StatusBar = False
    Application.ScreenUpdating = True

End Sub
```

Pour simplifier la discussion, j'ai créé une procédure DeleteXSurN qui supprime les X dernières lignes d'un groupe de N ligne sur la plage passée en argument.

Celle-ci va parcourir en sens inverse la plage par pas de N en supprimant les X dernière lignes, ce qui s'écrit :

```
Plage.Rows(compteur).Offset(N - X).Resize(X).Delete
```

Ensuite de quoi je vais copier les données vers le classeur macro.

Ce code va fonctionner parfaitement, mais il est extrêmement lent. Pour une suppression de 4 lignes sur 5 pour un fichier de 50 000 lignes, il mettra plus de deux minutes. Cela vient du fait que le processus de suppression d'Excel est lui-même un mécanisme lent. Parfois ce type de traitement est inévitable, mais souvent on peut contourner le problème.

Dans l'exercice qui nous préoccupe, il y a plusieurs façons d'accélérer le traitement, légèrement différent selon le résultat final que nous souhaitons obtenir. En effet, si comme dans mon exemple je souhaite être non destructif, c'est-à-dire que je n'enregistre pas les changements dans mon classeur source, alors le problème a été pris à l'envers. En effet, le but n'est dans ce cas pas de supprimer X lignes sur N, mais de récupérer N-X lignes sur N. En posant le problème ainsi, je peux utiliser plusieurs codes différents. Le plus simple serait alors d'écrire :

```
Public Sub TraiteFichier1()

Dim NomFichier As String, ClasseurSource As Workbook, Lignes As Range,
compteur As Long
Dim lgTime As Long

    NomFichier = Application.GetOpenFilename("Fichier Morphee (*.0*),
*.0*", , "Fichier d'acquisition")
    If StrComp(NomFichier, "Faux", vbTextCompare) = 0 Then Exit Sub
    Application.Workbooks.OpenText Filename:=NomFichier, StartRow:=2,
DataType:=xlDelimited, Tab:=True, DecimalSeparator:="."
    Set ClasseurSource = Application.ActiveWorkbook
    For compteur = 2 To ClasseurSource.Worksheets(1).UsedRange.Rows.Count
Step 5
        ClasseurSource.Worksheets(1).UsedRange.Rows(compteur).Copy
Destination:=ThisWorkbook.Worksheets("Donnee").Cells((compteur - 2) / 5 +
1, 1)
    Next
    ClasseurSource.Close False

End Sub
```

Dans ce code, nous ne lisons qu'une ligne sur 5 que nous intégrons directement dans la feuille donnée du classeur macro. Ce code va environ deux fois plus vite que le précédent, ce qui est bien mais nous pouvons encore largement améliorer. En effet, le code précédent est encore assez lent car nous utilisons un autre processus assez lourd, le copier coller. Or dans notre cas, nous voulons affecter les valeurs du fichier source dans le classeur macro, et pour cela, nous pouvons utiliser l'affectation directe des propriétés Value comme nous l'avons vu précédemment. Nous pourrions écrire :

```
Public Sub TraiteFichier2()

Dim NomFichier As String, ClasseurSource As Workbook, CellSource As Range,
CellCible As Range, compteur As Long, NbCol As Integer

    NomFichier = Application.GetOpenFilename("Fichier Morphee (*.0*),
*.0*", , "Fichier d'acquisition")
    If StrComp(NomFichier, "Faux", vbTextCompare) = 0 Then Exit Sub
    Application.Workbooks.OpenText Filename:=NomFichier, StartRow:=2,
DataType:=xlDelimited, Tab:=True, DecimalSeparator:="."
    Set ClasseurSource = Application.ActiveWorkbook
    Set CellSource = ClasseurSource.Worksheets(1).Cells(2, 1)
    Set CellCible = ThisWorkbook.Worksheets("Donnee").Cells(1, 1)
    NbCol = ClasseurSource.Worksheets(1).UsedRange.Columns.Count
    For compteur = 0 To ClasseurSource.Worksheets(1).UsedRange.Rows.Count \
5
        CellCible.Offset(compteur).Resize(, NbCol).Value =
CellSource.Offset(compteur * 5).Resize(, NbCol).Value
    Next
    ClasseurSource.Close False

End Sub
```

Pour traiter le même fichier de 50 000 lignes, ce code mettra moins de 5 secondes, soit une exécution 25 fois plus rapide que la première solution envisagée.

Cependant cette approche ne résout pas forcément tous les problèmes. On pourrait parfaitement imaginer que certains fichiers possèdent plus de 65536 lignes, ou vouloir écraser le fichier source avec un fichier ne contenant plus qu'une ligne sur 5. Le code suivant va pouvoir lire un fichier ayant plus de ligne que le maximum de lignes Excel et va créer un fichier source réduit.

```
Public Sub TraiteFichier3()

Dim NomFichier As String, FichierReduit As String, CellCible As Range,
compteur As Long, Recup As String

    NomFichier = Application.GetOpenFilename("Fichier Morphee (*.0*),
*.0*", , "Fichier d'acquisition")
    If StrComp(NomFichier, "Faux", vbTextCompare) = 0 Then Exit Sub
    FichierReduit = Left(NomFichier, InStrRev(NomFichier, ".") - 1) &
".red"
    Open NomFichier For Input As #1
    Open FichierReduit For Output As #2
    Line Input #1, Recup
    Line Input #1, Recup
    Set CellCible = ThisWorkbook.Worksheets("Donnee").Cells(1, 1)
    Do Until EOF(1)
        Line Input #1, Recup
        CellCible.Offset(compteur).Value = Recup
        Print #2, Recup
        compteur = compteur + 1
        Line Input #1, Recup
        Line Input #1, Recup
        Line Input #1, Recup
        Line Input #1, Recup
    Loop
    Close #1
    Close #2

CellCible.Resize(ThisWorkbook.Worksheets("Donnee").Rows.Count).TextToColumn
s Destination:=Range("A1"), DataType:=xlDelimited, Tab:=True

End Sub
```

Bien que ce code soit plus "universel" que le précédent, il n'en est pas moins plus rapide.

Comme nous venons de le voir, il n'est pas toujours évident de savoir quelle méthode choisir.

Recherche de plage

Il s'agit là d'un grand classique de la programmation Excel, la recherche des plages de données. Lorsqu'on travaille sur des fichiers externes, il est assez fréquent qu'on ne connaisse pas le nombre de lignes et ou de colonnes du fichier. Avant même de savoir quelle méthode choisir, nous devons déjà définir ce qu'est une plage de données. Nous allons continuer à travailler sur notre fichier d'acquisition, mais c'est fondamentalement la même chose pour tous les fichiers externes contenant des blocs de données.

Dans le cas d'un fichier d'acquisition, la plage de données c'est une matrice rectangulaire contenant tous les points d'acquisition. Généralement nous ne connaissons ni le nombre de ligne, ni le nombre de colonnes, mais uniquement les séparateurs permettant la redistribution.

La première question c'est comme pour le gruyère, avec ou sans trous. Soit on considère qu'il ne peut pas y avoir de cellules vides dans la plage, soit il peut y en avoir.

Excel propose des méthodes pour chercher la dernière cellule d'une plage dans les deux cas. Commençons par le plus simple, le cas sans trou.

Pour pouvoir définir une plage de données, il faut évidemment pouvoir atteindre facilement une des cellules de la plage. Dans nos fichiers d'acquisition, chaque colonne de données à un titre, il est donc assez simple d'aller rechercher le titre. Ensuite de quoi il va falloir aller chercher la dernière cellule de la colonne, c'est-à-dire la dernière cellule non vide.

```
Public Sub CherchePlage1()

Dim NomFichier As String, FeuilleSource As Worksheet, Plage As Range

    NomFichier = Application.GetOpenFilename("Fichier Morphee (*.0*),
*.0*", , "Fichier d'acquisition")
    If StrComp(NomFichier, "Faux", vbTextCompare) = 0 Then Exit Sub
    Application.Workbooks.OpenText Filename:=NomFichier, StartRow:=2,
DataType:=xlDelimited, Tab:=True, DecimalSeparator:="."
    Set FeuilleSource = Application.ActiveWorkbook.Worksheets(1)
    With FeuilleSource
        Set Plage = .Cells.Find("N_INS")
        If Not Plage Is Nothing Then
            Set Plage = .Range(Plage, Plage.End(xlDown))
        Else
            Set Plage = Application.InputBox("Sélectionnez la plage du
régime", Type:=8)
        End If
    End With
    MsgBox Plage.Address

End Sub
```

Jusque là aucune difficulté apparente, sauf s'il existe une cellule contenant le texte N_INS mais que les cellules de la même colonne sont vides. Il suffit de tester la cellule immédiatement au dessous pour avoir une approche 'universelle'. Si la plage est sensée ne pas contenir de formules, comme dans le cas d'un fichier d'acquisition, on peut aussi restreindre la plage à l'aide de SpecialCells comme dans l'exemple ci-dessous :

```
Public Sub CherchePlage1()

Dim NomFichier As String, FeuilleSource As Worksheet, Plage As Range

    NomFichier = Application.GetOpenFilename("Fichier Morphee (*.0*),
*.0*", , "Fichier d'acquisition")
    If StrComp(NomFichier, "Faux", vbTextCompare) = 0 Then Exit Sub
    Application.Workbooks.OpenText Filename:=NomFichier, StartRow:=2,
DataType:=xlDelimited, Tab:=True, DecimalSeparator:="."
    Set FeuilleSource = Application.ActiveWorkbook.Worksheets(1)
    With FeuilleSource
        Set Plage = .Cells.Find("N_INS")
        If Not Plage Is Nothing Then
            Set Plage = .Range(Plage,
Plage.End(xlDown)).SpecialCells(xlCellTypeConstants)
        Else
            Set Plage = Application.InputBox("Sélectionnez la plage du
régime", Type:=8)
        End If
    End With
    MsgBox Plage.Address

End Sub
```

On peut évidemment récupérer sur un principe assez similaire des plages de plusieurs colonnes. Cela fonctionne aussi pour les plages en lignes, il suffit de changer l'argument de la propriété End.

De la même façon, on peut aussi construire des plages discontinues à l'aide de la méthode Union, soit en connaissant la valeur du décalage d'une plage par rapport à une plage déterminée, soit par recherche successive. Par exemple, je peux construire une plage qui renverra les colonnes heure, N_INS, COUPLE_INS et PAP_INS quelle que soit leur position dans la feuille.

```
Public Sub PlageDiscontinue()

Dim NomFichier As String, NomColonne As String, compteur As Long
Dim FeuilleSource As Worksheet, tmpPlage As Range, DiscPlage As Range

    NomFichier = Application.GetOpenFilename("Fichier Morphee (*.0*),
*.0*", , "Fichier d'acquisition")
    If StrComp(NomFichier, "Faux", vbTextCompare) = 0 Then Exit Sub
    Application.Workbooks.OpenText Filename:=NomFichier, StartRow:=2,
DataType:=xlDelimited, Tab:=True, DecimalSeparator:="."
    Set FeuilleSource = Application.ActiveWorkbook.Worksheets(1)
    With FeuilleSource
        For compteur = 1 To 4
            NomColonne = Choose(compteur, "heure", "n_ins", "couple_ins",
"pap_ins")
            Set tmpPlage = .Cells.Find(What:=NomColonne, MatchCase:=False)
            If Not tmpPlage Is Nothing Then
                Set tmpPlage = .Range(tmpPlage,
tmpPlage.End(xlDown)).SpecialCells(xlCellTypeConstants)
                If Not DiscPlage Is Nothing Then
                    Set DiscPlage = Application.Union(DiscPlage, tmpPlage)
                Else
                    Set DiscPlage = tmpPlage
                End If
            Else
                MsgBox "Colonne " & UCase(NomColonne) & " non trouvée"
            End If
        Next compteur
    End With
    MsgBox DiscPlage.Address

End Sub
```

Vous noterez que je n'ai pas abordé la pratique pourtant évidente qui consisterai à tester les cellules successivement dans une énumération jusqu'à atteindre une cellule vide. Une recherche qui serait similaire à :

```
    With FeuilleSource
        Set Plage = .Cells.Find("N_INS")
        For Each MaCell In Plage.EntireColumn.Cells
            If IsEmpty(MaCell.Value) Then
                Set Plage = .Range(Plage, MaCell.Offset(-1))
                Exit For
            End If
        Next
    End With
```

Déjà parce que ce code contient un Bug, dans le sens où le code va planter si la cellule contenant N_INS n'est pas dans la première ligne, mais plus généralement parce que les codes d'énumérations sont extrêmement lents. Je n'utiliserai pas non plus de boucles For…Next classique pour les mêmes problèmes de lenteur.

Attaquons maintenant le cas un peu plus complexe des plages à trous éventuels, c'est-à-dire pouvant contenir des cellules vides. Dans ce cas il n'est plus possible d'utiliser la méthode End en défilant vers le bas (ou vers la droite pour les lignes), car End va s'arrêter à la cellule précédant la première cellule vide et la plage ne contiendra pas les valeurs suivantes.

Par contre il est assez simple de partir vers le bas pour aller chercher la première cellule non vide en remontant. Pour que cette approche fonctionne à tout coup, il faut atteindre la dernière cellule de la colonne (ou de la ligne) tester si elle est vide et appeler End vers le haut (ou vers la gauche) le cas échéant. Par exemple :

```
Public Sub CherchePlage2()

Dim NomFichier As String, FeuilleSource As Worksheet
Dim FirstCell As Range, LastCell As Range, Plage As Range

    Set FeuilleSource = ThisWorkbook.Worksheets(1)
    With FeuilleSource
        Set FirstCell = .Cells(1, 5)
        Set LastCell = .Cells(.Rows.Count, 5)
        If IsEmpty(LastCell.Value) Then
            Set LastCell = LastCell.End(xlUp)
        End If
        Set Plage = .Range(FirstCell, LastCell)
    End With
    MsgBox Plage.Address

End Sub
```

L'avantage de cette technique est qu'elle fonctionne dans tous les cas, sauf si la plage n'est pas délimitée par des cellules vides.

Recherche de valeur

La recherche de valeur peut sembler assez triviale dans l'absolu, pourtant elle peut vite se révéler problématique selon les cas. En effet, en première analyse, on a toujours tendance à penser que la méthode Find correctement utilisée répond à tous les problèmes, or tel n'est pas le cas. La méthode Find présente plusieurs inconvénients.

- Elle peut être sensible au type de données, ainsi un code de recherche de date peut échouer si la date est sous forme de chaîne ou sous forme de date
- Elle peut être limitée si la valeur cherchée apparaît de nombreuses fois, selon la forme du résultat qu'on souhaite renvoyer.
- Elle est souvent assez lente.

Il ne faut pas pour autant la mépriser, elle répond correctement à la plupart des cas, encore faut il la manipuler correctement.

Pour voir les diverses solutions envisageables, repartons d'un exemple classique.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Date	Heure	T°1	T°2	T°3	T°4	T°5	T°6	T°7	T°8	T°9	T°10
2	03/11/2002	14:40:16	19.3	40.6	21.5	37.4	40.7	18.7	40.5	18.3	18.4	238.8
3	03/11/2002	14:40:17	19.2	40.7	21.5	37.5	40.8	18.7	40.6	18.4	18.5	239.2
4	03/11/2002	14:40:18	19.2	41.2	21.5	37.6	41	18.7	40.6	18.4	18.6	239.7
5	03/11/2002	14:40:19	19.2	41.1	21.5	37.8	41.2	18.7	40.7	18.5	18.6	240.5
6	03/11/2002	14:40:20	19.3	41.5	21.5	37.9	41.3	18.7	40.7	18.6	18.7	241.2
7	03/11/2002	14:40:21	19.3	41.2	21.5	38.1	41.4	18.7	40.8	18.7	18.8	241.7
8	03/11/2002	14:40:22	19.3	41.8	21.5	38.2	41.6	18.6	40.8	18.7	18.8	242.2
9	03/11/2002	14:40:23	19.3	41.5	21.5	38.4	41.8	18.6	40.9	18.8	18.9	242.7
10	03/11/2002	14:40:24	19.3	42.3	21.5	38.4	41.9	18.6	40.9	19	19	243.4
11	03/11/2002	14:40:25	19.4	42.4	21.5	38.6	42.1	18.7	41	19	19.1	243.9
12	03/11/2002	14:40:26	19.4	42.5	21.5	38.8	42.2	18.6	41	19.1	19.1	244.4
13	03/11/2002	14:40:27	19.4	43	21.5	38.9	42.3	18.7	41.2	19.2	19.1	244.9
14	03/11/2002	14:40:28	19.4	43.2	21.5	39	42.5	18.6	41.2	19.2	19.2	245.3
15	03/11/2002	14:40:29	19.5	43.3	21.4	39.2	42.7	18.6	41.3	19.3	19.3	245.6
16	03/11/2002	14:40:30	19.5	43	21.5	39.3	42.8	18.6	41.3	19.5	19.3	246.1
17	03/11/2002	14:40:31	19.5	43.4	21.5	39.5	43	18.6	41.4	19.5	19.4	246.6
18	03/11/2002	14:40:32	19.5	43.7	21.5	39.6	43	18.6	41.4	19.6	19.4	246.8
19	03/11/2002	14:40:33	19.5	43.7	21.5	39.7	43.2	18.6	41.5	19.8	19.5	247.5
20	03/11/2002	14:40:34	19.5	43.9	21.5	39.9	43.3	18.6	41.5	19.8	19.6	248
21	03/11/2002	14:40:35	19.6	44	21.5	40	43.4	18.6	41.6	19.9	19.6	248.8
22	03/11/2002	14:40:36	19.6	44.3	21.5	40.1	43.6	18.6	41.5	20	19.7	249.2
23	03/11/2002	14:40:37	19.6	44.8	21.4	40.2	43.8	18.6	41.6	20.1	19.8	249.5
24	03/11/2002	14:40:38	19.6	44.7	21.5	40.3	43.9	18.6	41.7	20.1	19.9	250
25	03/11/2002	14:40:39	19.6	44.9	21.5	40.4	44.1	18.6	41.8	20.2	20	250.2
26	03/11/2002	14:40:40	19.7	44.9	21.4	40.6	44.2	18.6	41.8	20.3	20	250.7
27	03/11/2002	14:40:41	19.7	45.3	21.5	40.7	44.3	18.6	41.8	20.4	20.1	251.2
28	03/11/2002	14:40:42	19.7	45.7	21.5	40.9	44.5	18.6	41.9	20.4	20.2	251.4
29	03/11/2002	14:40:43	19.8	45.8	21.4	41.1	44.7	18.6	41.9	20.5	20.3	252.4
30	03/11/2002	14:40:44	19.7	46.2	21.4	41.2	44.7	18.6	41.9	20.6	20.3	252.9
31	03/11/2002	14:40:45	19.8	46.5	21.4	41.3	44.9	18.6	42	20.7	20.4	253.2
32	03/11/2002	14:40:46	19.8	46.4	21.5	41.5	45	18.6	42.1	20.8	20.5	253.4
33	03/11/2002	14:40:47	19.8	46.5	21.4	41.6	45.2	18.6	42.1	20.9	20.5	254.1
34	03/11/2002	14:40:48	19.8	47	21.4	41.7	45.3	18.6	42.1	21	20.6	254.4
35	03/11/2002	14:40:49	19.8	46.7	21.5	41.9	45.5	18.5	42.2	21.1	20.7	254.9
36	03/11/2002	14:40:50	19.9	47.1	21.4	42	45.6	18.6	42.2	21.2	20.8	255.3
37	03/11/2002	14:40:51	19.8	47.5	21.5	42.1	45.7	18.5	42.2	21.2	20.8	255.8

Feuil1 Feuil2 Feuil3

Il n'y a pas de sens aux exemples que je vais prendre, ceux-ci sont justes des exercices de manipulations.

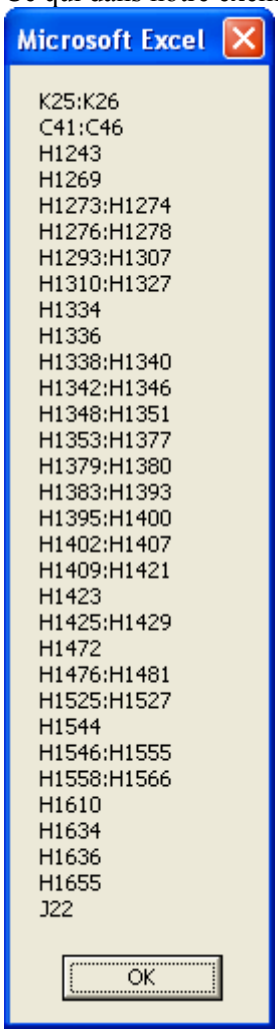
Nous voulons récupérer l'ensemble des cellules de la plage de données qui contiennent la valeur 20.

La première chose à faire généralement est de savoir si celle-ci existe au moins une fois dans la plage de recherche. Ensuite de quoi nous allons chercher à regrouper toutes les cellules qui contiennent cette valeur dans un objet Range. Nous utiliserons pour cela une fonction la plus universelle possible.

```
Public Sub TestRecherche()
    Dim MaPlage As Range, PlageReponse As Range, Zone As Range, Valeur As
Variant
    Dim Message As String

    Valeur = 20
    Set MaPlage = ThisWorkbook.Worksheets(1).UsedRange
    Set PlageReponse = PlageValeur(20, MaPlage)
    If Not PlageReponse Is Nothing Then
        For Each Zone In PlageReponse.Areas
            Message = Message & Zone.Address(False, False) & vbNewLine
        Next
    Else
        Message = "La valeur " & Valeur & " n'existe pas dans la plage " &
MaPlage.Address(False, False)
    End If
    MsgBox Message

End Sub
```

```
Public Function PlageValeur(ByVal ValeurCherchee As Variant, ByVal
PlageRecherche As Range) As Range

    Dim TrouveCell As Range, PremAdresse As String

    If Application.WorksheetFunction.CountIf(PlageRecherche,
ValeurCherchee) = 0 Then Exit Function
    Set TrouveCell = PlageRecherche.Find(What:=ValeurCherchee,
LookAt:=xlWhole, MatchCase:=False)
    PremAdresse = TrouveCell.Address
    Set PlageValeur = TrouveCell
    Do
        Set TrouveCell = PlageRecherche.FindNext(TrouveCell)
        Set PlageValeur = Application.Union(PlageValeur, TrouveCell)
    Loop Until TrouveCell.Address = PremAdresse

End Function
```

Ce qui dans notre exemple renverra :



Ce code est relativement efficace sauf si la valeur apparaît dans plusieurs milliers de cellules.

Nous pourrions cependant concevoir une autre approche assez différente, mais nettement plus rapide travaillant elle par l'élimination de cellules. En effet, la méthode ColumnDifferences par exemple permet de trouver toutes les cellules d'une colonne ne contenant pas la valeur de la cellule désignée. Or par exclusion, cela revient au même que la recherche de valeur. Nous pourrions donc écrire une fonction telle que :

```
Public Function PlageValeur1(ByVal ValeurCherchee As Variant, ByVal
PlageRecherche As Range) As Range
```

```
    Dim Colonne As Range, TrouveCell As Range, NotEqual As Range

    If Application.WorksheetFunction.CountIf(PlageRecherche,
ValeurCherchee) = 0 Then Exit Function
    For Each Colonne In PlageRecherche.Columns
        If Application.WorksheetFunction.CountIf(Colonne, ValeurCherchee) >
0 Then
            Set TrouveCell = Colonne.Find(What:=ValeurCherchee,
LookAt:=xlWhole, MatchCase:=False)
            Set NotEqual = Colonne.ColumnDifferences(TrouveCell)
            NotEqual.EntireRow.Hidden = True
            If PlageValeur1 Is Nothing Then
                Set PlageValeur1 = Colonne.SpecialCells(xlCellTypeVisible)
            Else
                Set PlageValeur1 = Application.Union(PlageValeur1,
Colonne.SpecialCells(xlCellTypeVisible))
            End If
            NotEqual.EntireRow.Hidden = False
        End If
    Next Colonne

End Function
```

Evidemment ce code est un peu étrange. Comment fonctionne-t-il ?

Je vais travailler en colonne, mais dans l'absolu il faudrait détecter la dimension la moins grande pour travailler dans ce sens. Le code parcours les colonnes de la plage de recherche, à chaque colonne il vérifie s'il existe au moins une fois la valeur cherchée dans la colonne. Si tel est le cas, il appelle la méthode ColumnDifferences qui renvoie la plage de toutes les cellules ne contenant pas la valeur cherchée. Il masque alors cette plage et récupère la plage de cellules visible qui par la force des choses contiennent la valeur cherchée. Il restaure ensuite la visibilité des cellules. Bien que curieuse, cette méthode de recherche est extrêmement rapide.

Ces deux méthodes n'en ont pas moins un inconvénient, elles ne fonctionnent que pour la recherche d'une valeur. Si on recherche plusieurs valeurs, une plage de valeur, une inégalité et ainsi de suite, ça ne marche plus.

Nous allons donc travailler avec une méthode beaucoup plus puissante qui permet de gérer de nombreux cas de recherche, le filtrage. Vous allez voir que dans le principe, c'est la même chose que le code précédent.

Il existe une méthode de filtrage avancée beaucoup plus puissante que nous n'utiliserons pas ici car elle est un peu complexe, nous nous contenterons d'utiliser la méthode AutoFilter qui déjà devrait répondre à bien des attentes.

Commençons par le cas identique de la recherche de valeur.

```
Public Function PlageValeur2(ByVal ValeurCherchee As Variant, ByVal
PlageRecherche As Range) As Range

    Dim Colonne As Range, TrouveCell As Range, NotEqual As Range

    If Application.WorksheetFunction.CountIf(PlageRecherche,
ValeurCherchee) = 0 Then Exit Function
    For Each Colonne In PlageRecherche.Columns
        If Application.WorksheetFunction.CountIf(Colonne, ValeurCherchee) >
0 Then
            Colonne.AutoFilter 1, ValeurCherchee
            'si ligne de titre
            PlageRecherche.Cells(1).EntireRow.Hidden = True
            If PlageValeur2 Is Nothing Then
                Set PlageValeur2 = Colonne.SpecialCells(xlCellTypeVisible)
            Else
                Set PlageValeur2 = Application.Union(PlageValeur2,
Colonne.SpecialCells(xlCellTypeVisible))
            End If
            Colonne.AutoFilter
            'si ligne de titre
            PlageRecherche.Cells(1).EntireRow.Hidden = False
        End If
    Next Colonne

End Function
```

A part le fait de masquer la ligne de titre pour ne pas comptabiliser de mauvaises cellules, ce code est tout à fait similaire au précédent. L'avantage de cette technique est que sans modification, elle permet des recherches beaucoup plus évoluées. En effet, pour rechercher les valeurs supérieures à 20, il suffit de passer comme argument ">20" à la fonction. On peut d'ailleurs aller plus loin puisqu'on peut aussi passer un critère de recherche de chaîne du type "*Ins" pour toutes les chaînes finissant par 'ins'.

Autres recherches

Pour finir avec les recherches, nous allons nous livrer à un petit exercice de style amusant, pour voir en quoi la conception d'un programme dépend de la façon de conceptualiser les problèmes mais aussi de la bonne connaissance du modèle objet.

Commençons par écrire le code suivant :

```
Public Sub DisperseStyle()

    Dim MinRowCol As Long, MaxCol As Long, MaxRow As Long
    Dim NumCol As Long, NumRow As Long, compteur As Long

    MinRowCol = 1
    MaxCol = 256
    MaxRow = 65536
    Randomize
    For compteur = 1 To 100
        NumCol = Int((MaxCol - MinRowCol + 1) * Rnd) + MinRowCol
        NumRow = Int((MaxRow - MinRowCol + 1) * Rnd) + MinRowCol
        ThisWorkbook.Worksheets(1).Cells(NumRow, NumCol).Font.Bold = True
    Next compteur

End Sub
```

La fonction Randomize active le générateur de nombre aléatoire.

Ce code va mettre en gras la police de 100 cellules aléatoirement dans la feuille.

Maintenant nous allons devoir écrire une procédure qui retrouve ses cellules. L'approche formelle est assez simple, il suffit de parcourir l'ensemble des cellules et de tester la police de caractère. Par exemple :

```
Public Sub RechercheStyle1()

Dim MaCell As Range, Plage As Range, MaFeuille As Worksheet

    Set MaFeuille = ThisWorkbook.Worksheets(1)
    For Each MaCell In MaFeuille.Cells
        If MaCell.Font.Bold = True Then
            If Plage Is Nothing Then
                Set Plage = MaCell
            Else
                Set Plage = Application.Union(Plage, MaCell)
            End If
        End If
    Next MaCell
    Plage.Select

End Sub
```

Ce code est indiscutable dans son fonctionnement si ce n'est sur un point, il est abominablement lent. Sur un poste assez récent, il lui faudra plus de trois minutes pour renvoyer la plage des 100 cellules.

Est-il possible d'accélérer ce code, et si oui comment ?

Si vous vous rappelez ce que nous avons dit pour les propriétés de l'objet Range, vous avez peut être la solution. En effet, une plage renvoie la valeur de la propriété si elle est commune à toutes les cellules de l'objet Range et une valeur particulière si non. Ce qui veut dire que si je parcours l'ensemble des lignes de la feuille, la ligne renverra False si aucune des cellules n'a sa police en gras et Null si elle contient au moins une cellule en Gras. En appliquant le même raisonnement sur les colonnes, je vais pouvoir réduire énormément le nombre de tests à effectuer. Examinons le code suivant :

```
Public Sub RechercheStyle2()

Dim PlageReduite As Range, MaFeuille As Worksheet, Plage As Range
Dim EnumPlage As Range, MaCell As Range

    Set MaFeuille = ThisWorkbook.Worksheets(1)
    For Each EnumPlage In MaFeuille.Columns
        If IsNull(EnumPlage.Font.Bold) Then
            If PlageReduite Is Nothing Then
                Set PlageReduite = EnumPlage
            Else
                Set PlageReduite = Application.Union(PlageReduite,
EnumPlage)
            End If
        End If
    Next
    For Each EnumPlage In MaFeuille.Rows
        If IsNull(EnumPlage.Font.Bold) Then
            For Each MaCell In Application.Intersect(EnumPlage,
PlageReduite)
                If MaCell.Font.Bold Then
                    If Plage Is Nothing Then
                        Set Plage = MaCell
                    Else
                        Set Plage = Application.Union(Plage, MaCell)
                    End If
                End If
            Next MaCell
```

```
        End If
    Next EnumPlage
    Plage.Select

End Sub
```

Nous allons donc parcourir les colonnes de la feuille et mettre toutes celles qui renvoient Null dans un objet Range. Nous allons ensuite parcourir la collection des lignes de la feuille. Chaque fois qu'une ligne va renvoyer Null, elle contiendra au moins une cellule dont la police est en gras. Cette ou ces cellules seront forcément à l'intersection de la ligne et de la plage réduite des colonnes. Il suffit donc de parcourir une à une les cellules de cette intersection et de récupérer celles qui répondent au critère.

Ce code est nettement plus complexe, mais il est 200 fois plus rapide que le précédent. Sur le même poste, il met moins de 2 secondes pour renvoyer la plage.

Fonctions de feuille de calcul

Dans certain cas, il est intéressant de pouvoir écrire des fonctions utilisables dans des cellules de feuille de calcul, comme pour les fonctions intégrées d'Excel.

Par convention, on définit toujours une fonction de type Variant pour ce type de fonctions. Sauf dans certains cas particulier, on marque la fonction comme étant volatile, c'est-à-dire devant être recalculée chaque fois qu'Excel calcul les formules du classeur. On utilise pour cela la méthode *Volatile* de l'objet *Application*.

Il revient au développeur de gérer les erreurs propres à la fonction et de renvoyer une valeur d'erreur dans la cellule appelante. Prenons l'exemple suivant qui compte le nombre de cellules contenant une valeur numérique dans la plage passée en argument.

```
Public Function CompteNombre(ByVal Plage As Range) As Variant

    Application.Volatile
    If Not TypeOf Plage Is Range Then
        CompteNombre = CVErr(xlErrValue)
    Else
        Dim Cellule As Range
        For Each Cellule In PlageAr.Cells
            If IsNumeric(Cellule.Value) Then CompteNombre = CompteNombre +
1
        Next Cellule
    End If

End Fonction
```

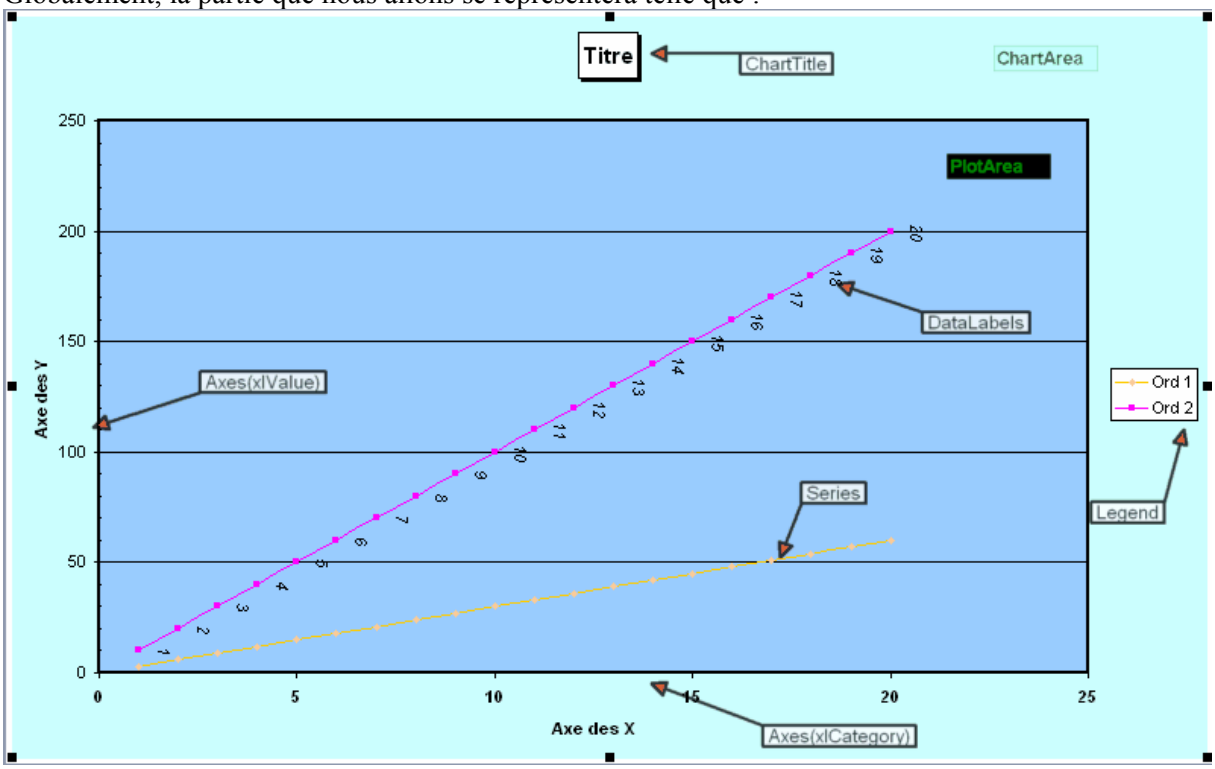
Notez que le test TypeOf n'est là que pour vous donnez un exemple du renvoi d'erreur, dans la réalité Excel n'appellera pas la fonction si l'argument n'est pas du bon type.

Manipulation des graphiques

Les graphiques d'Excel sont aussi présents dans le modèle objet. Il n'existe qu'un objet graphique, nommé Chart, bien qu'il y en ait deux formes :

- La feuille graphique qui renvoie un Objet Chart au travers de la collection Charts du classeur
- Le graphique incorporé (à une feuille de calcul) qui renvoie un objet Chart contenu dans un objet ChartObject renvoyé par la collection ChartObjects de la feuille de calcul.

Globalement, la partie que nous allons se représentera telle que :



Le modèle objet dépendant des graphiques est assez dense mais sa programmation reste assez simple. En fait elle se décompose en deux parties générales, la manipulation des données permettant de créer le graphique, et la mise en forme.

Créer un Graphique

Dans la création du graphique, on entend à minima, la création de l'objet graphe et la création d'au moins une série.

Pour créer une feuille graphique, nous aurons donc un code de la forme :

```
Classeur.Charts.Add
```

Pour créer un graphique incorporé, nous aurons un code de la forme :

```
Classeur.Feuille.ChartObjects.Add
```

Pour pouvoir manipuler facilement notre graphique, nous allons l'affecter à une variable objet de type Chart.

Pour créer une feuille graphique nous allons avoir :

```
Sub FeuilleGraphique()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range

Set MonGraphe = ThisWorkbook.Charts.Add

End Sub
```

Dans le cas du graphique incorporé c'est un peu différent. Lors de sa création dans la méthode Add, celle-ci attend quatre arguments définit tels que :

Function Add(*Left As Double, Top As Double, Width As Double, Height As Double*) **As ChartObject**

Vous devez donc définir des coordonnées Gauche et Haute ainsi qu'une largeur et une hauteur.

On peut soit passer des valeurs directement telles que :

```
Sub GraphiqueIncorpore()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range

Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
With MaFeuille
    Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1,
1).End(xlDown)).Resize(, 6)
    PlageDonnees.Select
    Set MonGraphe = .ChartObjects.Add(100, 100, 300, 200).Chart
End With

End Sub
```

Soit aligner l'objet sur une plage de cellules :

```
Sub GraphiqueIncorpore()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageGraphique As Range

Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
With MaFeuille
    Set PlageGraphique = .Range("H1:N20")
    Set MonGraphe = .ChartObjects.Add(PlageGraphique.Left,
PlageGraphique.Top, PlageGraphique.Width, PlageGraphique.Height).Chart
End With

End Sub
```

Dans les deux cas, vous voyez que ce n'est pas l'objet que je viens de créer que j'affecte à ma variable MonGraphe mais la valeur renvoyée par sa propriété Chart. En effet, la méthode Add de la collection ChartObjects renvoie un objet ChartObject qui lui-même contient l'objet Chart.

Une fois la variable MonGraphe correctement affectée, il n'y aura plus de différences de manipulation entre la feuille graphique ou le graphique incorporé, du moins au niveau de la programmation du graphe.

Nous allons donc créer des séries en partant de notre désormais classique tableau.

	A	B	C	D	E	F	G
1	Temps (s)	T° 1	T° 2	T° 3	T° 4	Vitesse	
2	0	40.6	21.5	37.4	40.7	2497	
3	1	40.7	21.5	37.5	40.8	2500	
4	2	41.2	21.5	37.6	41	2500	
5	3	41.1	21.5	37.8	41.2	2500	
6	4	41.5	21.5	37.9	41.3	2495	
7	5	41.2	21.5	38.1	41.4	2500	
8	6	41.8	21.5	38.2	41.6	2497	
9	7	41.5	21.5	38.4	41.8	2492	
10	8	42.3	21.5	38.4	41.9	2495	
11	9	42.4	21.5	38.6	42.1	2495	
12	10	42.5	21.5	38.8	42.2	2497	
13	11	43	21.5	38.9	42.3	2500	
14	12	43.2	21.5	39	42.5	2500	
15	13	43.3	21.4	39.2	42.7	2500	
16	14	43	21.5	39.3	42.8	2500	
17	15	43.4	21.5	39.5	43	2497	
18	16	43.7	21.5	39.6	43	2500	
19	17	43.7	21.5	39.7	43.2	2497	
20	18	43.9	21.5	39.9	43.3	2500	
21	19	44	21.5	40	43.4	2497	
22	20	44.3	21.5	40.1	43.6	2500	
23	21	44.8	21.4	40.2	43.8	2497	
24	22	44.7	21.5	40.3	43.9	2497	
25	23	44.9	21.5	40.4	44.1	2497	
26	24	44.9	21.4	40.6	44.2	2500	
27	25	45.3	21.5	40.7	44.3	2500	
28	26	45.7	21.5	40.9	44.5	2500	
29	27	45.8	21.4	41.1	44.7	2495	
30	28	46.2	21.4	41.2	44.7	2497	
31	29	46.5	21.4	41.3	44.9	2490	
32	30	46.4	21.5	41.5	45	2495	
33	31	46.5	21.4	41.6	45.2	2500	
34	32	47	21.4	41.7	45.3	2497	
35	33	46.7	21.5	41.9	45.5	2500	
36	34	47.1	21.4	42	45.6	2500	
37	35	47.5	21.5	42.1	45.7	2497	

Tableau | Feuil2

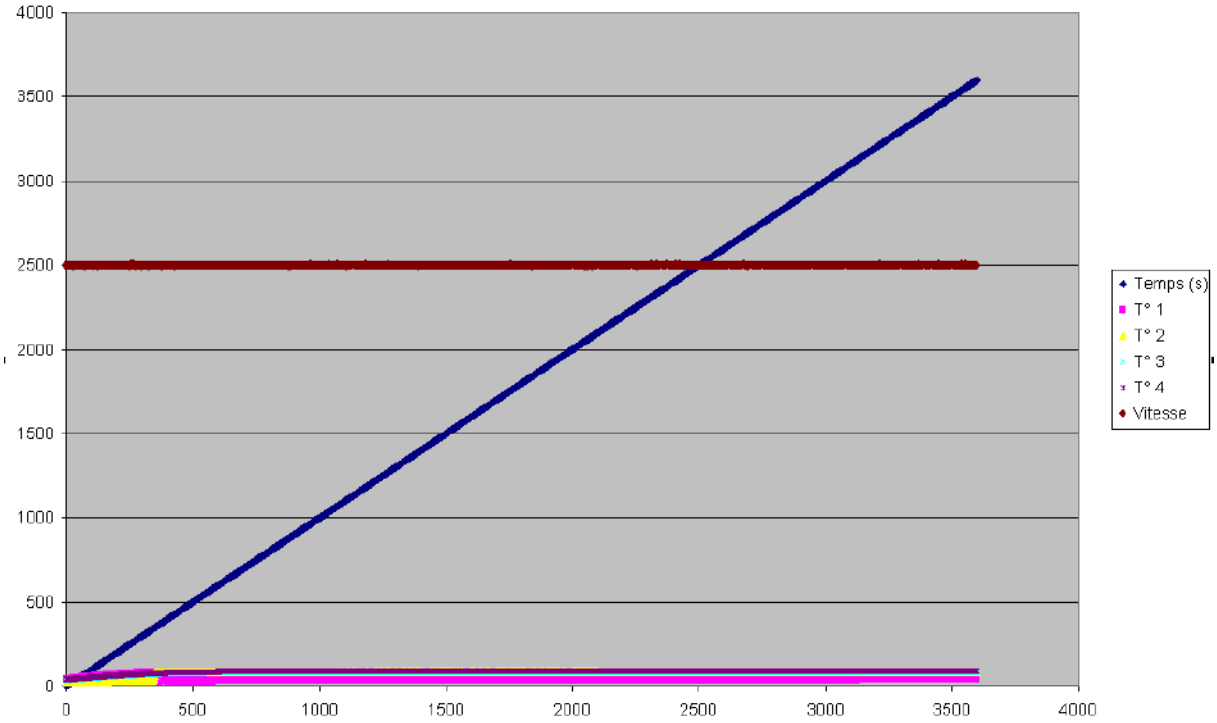
Dessin | Formes automatiques

Utiliser la sélection

La première méthode est un peu spécieuse, dans le sens ou elle engendre plus souvent des erreurs que de bons résultats. En effet, lorsque vous créez une feuille graphique alors qu'une plage de données est sélectionnée, Excel va utiliser cette sélection pour créer des séries. Imaginons le code suivant :

```
Sub CreeGrapheSelection()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range

    Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
    With MaFeuille
        Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1,
1).End(xlDown)).Resize(, 6)
        PlageDonnees.Select
    End With
    Set MonGraphe = ThisWorkbook.Charts.Add
    MonGraphe.ChartType = xlXYScatter

End Sub
```

Je sélectionne une plage contenant des données et je crée une feuille graphique. Celle-ci aura alors cette apparence :



La propriété ChartType permet de définir le type de graphique. Nous la verrons lors de la mise en forme des graphes. xlXYScatter défini un graphique X,Y appelé nuage de points.

Cette méthode fonctionne bien, mais si vous regardez la légende, vous verrez que le temps est considéré comme une série de données et non comme l'abscisse des autres séries. On évite généralement l'emploi de cette technique. Cependant cela doit vous donner un second réflexe. Comme Excel va toujours tenter de créer un graphe avec sa sélection, vous n'avez pas la certitude que l'objet graphique que vous venez de créer ne contiendra pas une série ajoutée par Excel.

Pour cela, on appelle la méthode Clear de l'objet Chart juste après sa création.

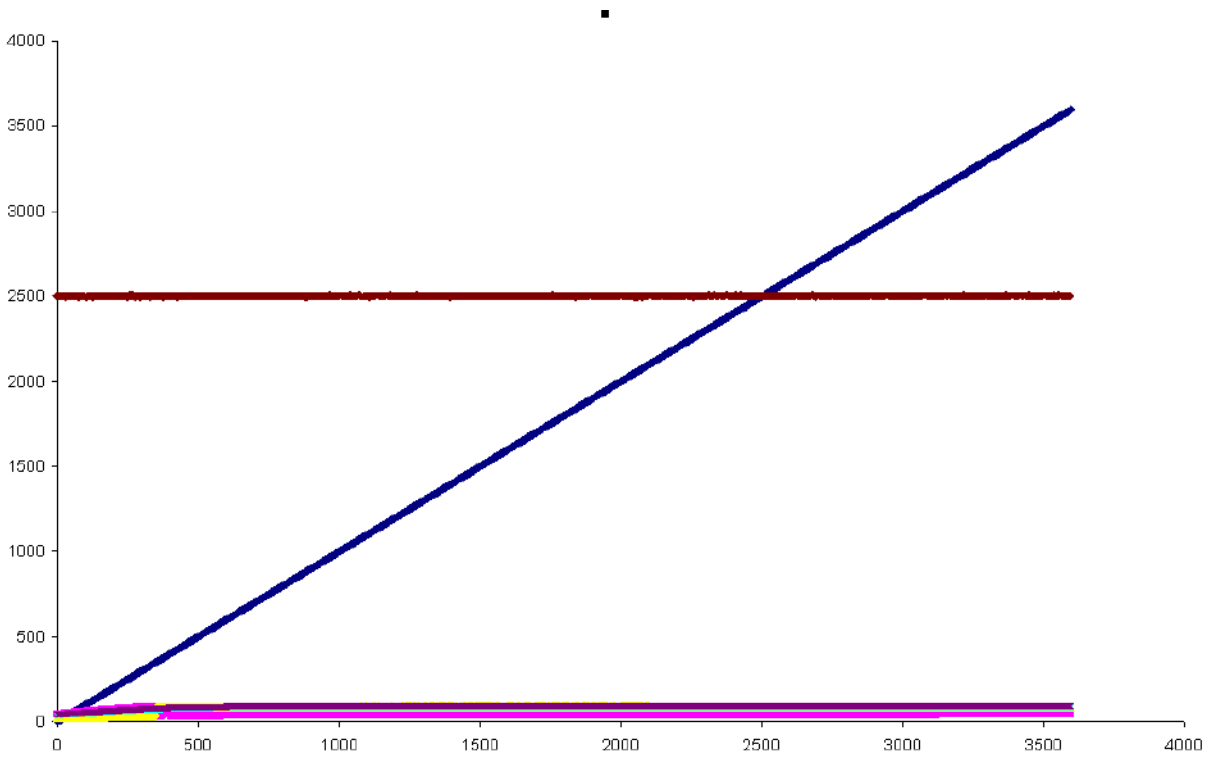
Création par Copier Coller

La deuxième méthode consiste à utiliser le copier coller, c'est-à-dire copier les cellules de la plage de données et les coller dans le graphique.

Donc nous aurons :

```
Sub CreeGrapheCopy()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range

    Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
    With MaFeuille
        Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1,
1).End(xlDown)).Resize(, 6)
    End With
    Set MonGraphe = ThisWorkbook.Charts.Add
    MonGraphe.ChartArea.Clear
    MonGraphe.ChartType = xlXYScatter
    PlageDonnees.Copy
    MonGraphe.Paste

End Sub
```

Qui nous créera le graphe :



Là encore, la colonne des abscisses n'est pas correctement interprétée. Cependant nous pourrions modifier ce comportement en faisant un collage non plus sur l'objet Chart mais sur la collection SeriesCollection, tel que :

```
Sub CreeGrapheCopy()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range

    Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
    With MaFeuille
        Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1,
1).End(xlDown)).Resize(, 6)
    End With
    Set MonGraphe = ThisWorkbook.Charts.Add
    MonGraphe.ChartArea.Clear
    MonGraphe.ChartType = xlXYScatter
    PlageDonnees.Copy
    MonGraphe.SeriesCollection.Paste Rowcol:=xlColumns, SeriesLabels:=True,
CategoryLabels:=True, Replace:=False, NewSeries:=True

End Sub
```

Dans ce cas, le graphique utilisera bien la colonne de gauche de la plage de données comme colonne des abscisses.

Définition d'une source de données

Nous pouvons aussi affecter la plage de données au graphique par le biais de la méthode SetSourceData. Celle-ci attend un objet Range définissant la plage de données et éventuellement un argument PlotBy définissant le sens des séries (en ligne ou en colonne)

```
Sub CreeGrapheDonneeSource()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range

    Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
    With MaFeuille
        Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1,
1).End(xlDown)).Resize(, 6)
    End With
    Set MonGraphe = ThisWorkbook.Charts.Add
    MonGraphe.ChartArea.Clear
    MonGraphe.ChartType = xlXYScatter
    MonGraphe.SetSourceData PlageDonnees

End Sub
```

Par ajout de séries

D'une manière assez similaire, nous pouvons aussi passer la plage de données comme argument de la méthode Add de l'objet SeriesCollection.

```
Sub CreeGrapheAddSerie()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range

    Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
    With MaFeuille
        Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1,
1).End(xlDown)).Resize(, 6)
    End With
    Set MonGraphe = ThisWorkbook.Charts.Add
    MonGraphe.ChartArea.Clear
    MonGraphe.ChartType = xlXYScatter
    MonGraphe.SeriesCollection.Add PlageDonnees

End Sub
```

Vous noterez également que sans autres arguments, cette technique prends en compte la première colonne comme une série indépendante et non comme des abscisses.

Vous pouvez changer cela en utilisant :

```
MonGraphe.SeriesCollection.Add PlageDonnees, xlColumns, , True
```

Par définitions des séries

Toutes ces méthodes sont pratiques, mais elles ne sont pas très souples. En effet, si vous voulez des abscisses qui ne soient pas dans la première colonne de la plage, vous ne pouvez pas les utiliser. La façon la plus robuste consiste à utiliser une vraie programmation des objets séries, un à un.

Reprenons la base de ce modèle objet. L'objet Chart expose une propriété SeriesCollection, qui renvoie la collection des séries du graphique.

Les membres de cette collection sont des objets Series. Un objet Series contient plusieurs propriétés et méthodes qui définissent l'ensemble de la série c'est-à-dire ses données et son apparence.

Pour les données, l'objet Series expose une propriété Values qui définie ces ordonnées, et une propriété XValues qui définit ces abscisses.

De fait, nous n'allons donc plus travailler avec toute la plage de colonnes, mais avec des colonnes séparées.

Recréons notre graphique précédent avec cette technique.

```
Sub CreeGrapheNewSerie()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range
Dim PlageX As Range, PlageY As Range, MaSerie As Series, compteur As Long

    Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
    With MaFeuille
        Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1,
1).End(xlDown)).Resize(, 6)
    End With
    Set MonGraphe = ThisWorkbook.Charts.Add
    MonGraphe.ChartArea.Clear
    MonGraphe.ChartType = xlXYScatter
    Set PlageX = PlageDonnees.Columns(1)
    For compteur = 1 To PlageDonnees.Columns.Count - 1
        Set PlageY = PlageX.Offset(, compteur)
        Set MaSerie = MonGraphe.SeriesCollection.NewSeries
        With MaSerie
            .Values = PlageY
            .XValues = PlageX
            .Name = PlageDonnees.Cells(1, 1).Offset(, compteur)
        End With
    Next compteur

End Sub
```

La propriété Name définit le nom qui apparaîtra par défaut dans la légende.

L'ajout de série à un graphique déjà existant fonctionne à l'identique, si ce n'est que vous n'avez pas besoin de créer l'objet graphique.

Mise en forme

Le code de mise en forme est très simple. Je ne vais pas entrer dans le détail du modèle objet mais juste vous donnez des exemples de codes classiques.

Modifier l'apparence des séries

Le code suivant va changer le marqueur de série et la couleur du tracé pour chaque série lors de sa création.

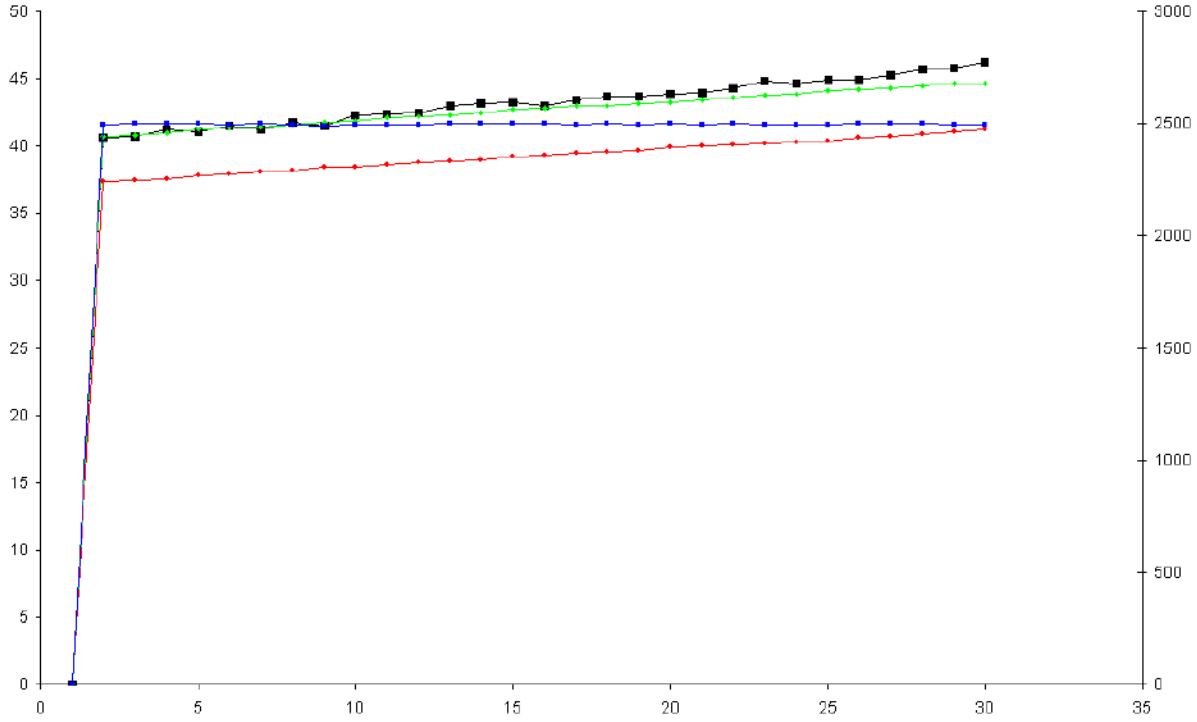
```
Sub CreeGrapheNewSerie()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range
Dim PlageX As Range, PlageY As Range, MaSerie As Series, compteur As Long

    Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
    With MaFeuille
        Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1,
1).End(xlDown)).Resize(30, 6)
    End With
    Set MonGraphe = ThisWorkbook.Charts.Add
    MonGraphe.ChartArea.Clear
    MonGraphe.ChartType = xlXYScatter
    Set PlageX = PlageDonnees.Columns(1)
    For compteur = 1 To PlageDonnees.Columns.Count - 1
        Set PlageY = PlageX.Offset(, compteur)
        Set MaSerie = MonGraphe.SeriesCollection.NewSeries
        With MaSerie
            .Values = PlageY
            .XValues = PlageX
            .Name = PlageDonnees.Cells(1, 1).Offset(, compteur)
            .Border.ColorIndex = compteur
```

```
            .MarkerStyle = Choose(compteur, xlMarkerStylePlus,
xlMarkerStyleTriangle, xlMarkerStyleCircle, xlMarkerStyleDiamond,
xlMarkerStyleSquare)
            .MarkerSize = 3
            .MarkerBackgroundColorIndex = compteur
            .MarkerForegroundColorIndex = compteur
        End With
    Next compteur

End Sub
```

Le code suivant affecte la série 'Vitesse' à un deuxième axe Y.

```
Public Sub ModifierSerie()

    Dim MonGraphe As Chart, MaSerie As Series

    Set MonGraphe = ThisWorkbook.Charts(1)
    For Each MaSerie In MonGraphe.SeriesCollection
        If InStr(1, MaSerie.Name, "vitesse", vbTextCompare) > 0 Then
            MaSerie.AxisGroup = xlSecondary
        End If
    Next MaSerie

End Sub
```

Ce code ajouter au précédent donnerait un graphe similaire à :



Comme vous le voyez, les séries partent d'un point '0' qui n'existe pas dans le tableau de données. Comment cela se fait-il ?

En fait, j'ai sélectionné comme plage de données des colonnes contenant un titre pour la série. Ce titre ne pouvant pas être converti en données, Excel trace un point 0 qui n'existe pas. Je pourrais corriger cela, soit en modifiant la plage de données, soit en modifiant la première valeur de la série.

Dans les deux cas, cette approche n'est pas facile car on ne peut pas trop modifier la collection Points de l'objet Serie. Par ailleurs, les propriétés Values et XValues de la série ne contiennent pas une référence vers l'objet Range qui les a créées mais un variant contenant un tableau des valeurs.

Pour retrouver les objets Range d'origine, je vais devoir manipuler une autre propriété de l'objet Serie, FormulaR1C1Local. Celle-ci va renvoyer une chaîne de type :

```
=SERIE("T° 1";Tableau!L1C1:L30C1;Tableau!L1C2:L30C2;1)
```

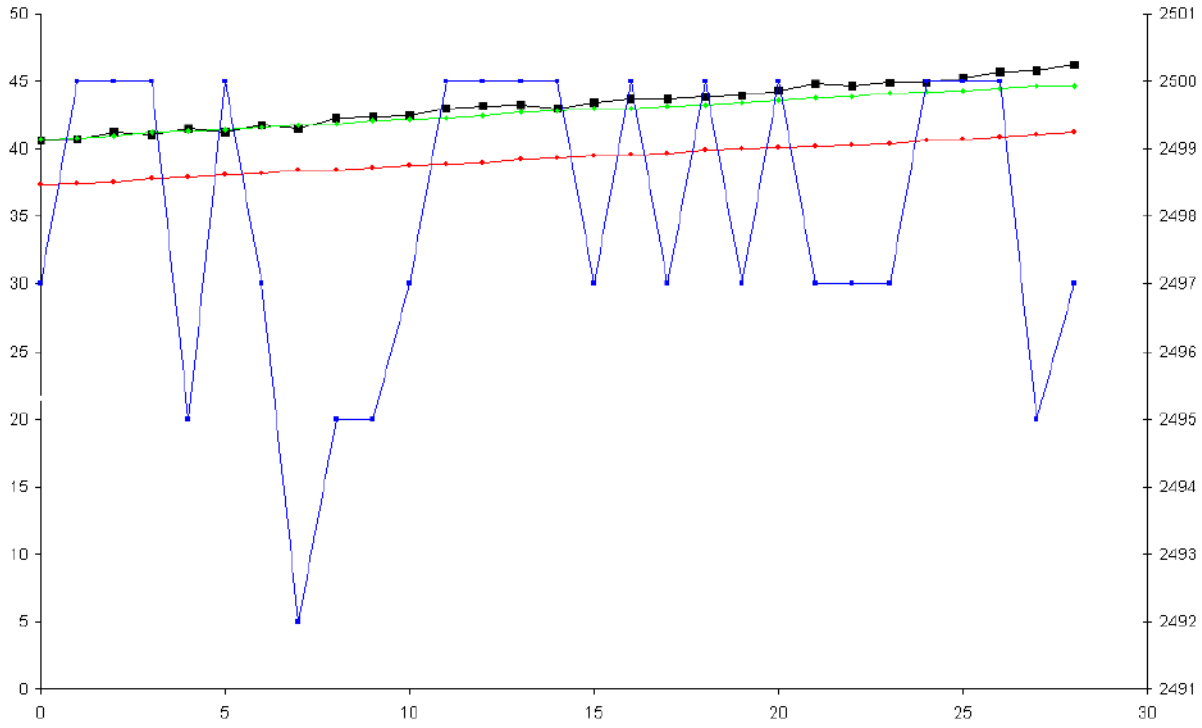
C'est-à-dire une fonction comprenant comme argument :

- Le Nom de la série
- La plage des abscisses
- La plage des ordonnées
- L'ordre de traçage

Il s'agit d'une chaîne de caractère, je peux donc la manipuler comme telle. Pour supprimer le premier point de chaque série, il suffirait par exemple d'écrire :

```
Public Sub ModifierSerie()

    Dim MonGraphe As Chart, MaSerie As Series

    Set MonGraphe = ThisWorkbook.Charts(1)
    For Each MaSerie In MonGraphe.SeriesCollection
        MaSerie.FormulaR1C1Local = Replace(MaSerie.FormulaR1C1Local, "L1C",
"L2C")
        If InStr(1, MaSerie.Name, "vitesse", vbTextCompare) > 0 Then
            MaSerie.AxisGroup = xlSecondary
        End If
    Next MaSerie

End Sub
```

Ce qui nous donnerait un graphe tel que :



Ajouter un titre ou un fond au graphique

Ce code va ajouter un fond à la zone extérieure du graphe, un autre à la zone intérieure, puis ajouter un titre encadré et ombré.

```
Public Sub TitreEtFond()

    Dim MonGraphe As Chart

    Set MonGraphe = ThisWorkbook.Charts(1)
    With MonGraphe
        With .ChartArea.Interior
            .ColorIndex = 34
            .PatternColorIndex = 2
            .Pattern = xlSolid
        End With
        With .PlotArea.Interior
            .ColorIndex = 15
            .PatternColorIndex = 1
            .Pattern = xlSolid
        End With
        .HasTitle = True
        With .ChartTitle
            .Text = "Titre"
            .Border.LineStyle = xlThin
            .Shadow = True
        End With
    End With

End Sub
```

Notez que je pourrais mettre une cellule comme Titre en écrivant :

```
.Text = "=Tableau!R13C11"
```

Manipuler la légende

Le code suivant ajoute une légende au graphique et met les éléments de la légende en gras.

```
Public Sub AjoutLegende()

    Dim MonGraphe As Chart, Element As LegendEntry

    Set MonGraphe = ThisWorkbook.Charts(1)
    With MonGraphe
        .HasLegend = True
        With .Legend
            .Position = xlLegendPositionRight
            For Each Element In .LegendEntries
                Element.Font.Bold = True
            Next Element
        End With
    End With

End Sub
```

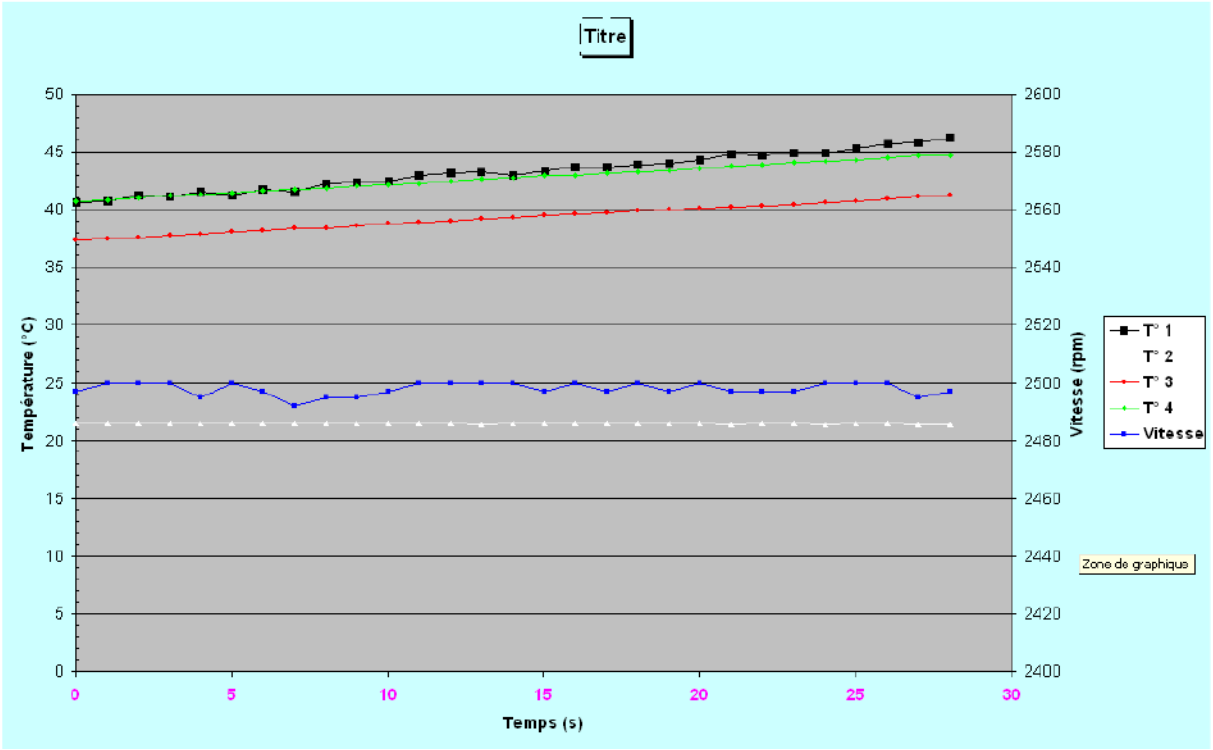
Manipuler les axes

Le code suivant va modifier les axes en :

- Ajoutant une légende aux trois axes
- Modifiant la police de l'axe des abscisses
- Modifiant les marqueurs de l'axe des ordonnées de gauche
- Définissant un quadrillage interne
- Modifiant l'échelle de l'axe des ordonnées de droite

```
Public Sub MiseEnFormeAxes()

    Dim MonGraphe As Chart, Axe As Axis

    Set MonGraphe = ThisWorkbook.Charts(1)
    With MonGraphe
        'actions sur l'axe des X
        Set Axe = .Axes(xlCategory, xlPrimary)
        With Axe
            .HasTitle = True
            .AxisTitle.Text = "Temps (s)"
            .TickLabels.Font.ColorIndex = 7
            .TickLabels.Font.Bold = True
        End With
        'actions sur les ordonnées de gauche
        Set Axe = .Axes(xlValue, xlPrimary)
        With Axe
            .HasTitle = True
            .AxisTitle.Text = "Température (°C)"
            .MajorTickMark = xlTickMarkOutside
            .MinorTickMark = xlTickMarkInside
            .HasMajorGridlines = True
        End With
        'actions sur les ordonnées de droite
        Set Axe = .Axes(xlValue, xlSecondary)
        With Axe
            .HasTitle = True
            .AxisTitle.Text = "Vitesse (rpm)"
            .MinimumScale = 2400
            .MaximumScale = 2600
        End With
    End With

End Sub
```

Tous ces codes enchaînés de mise en forme donneraient :



Débogage

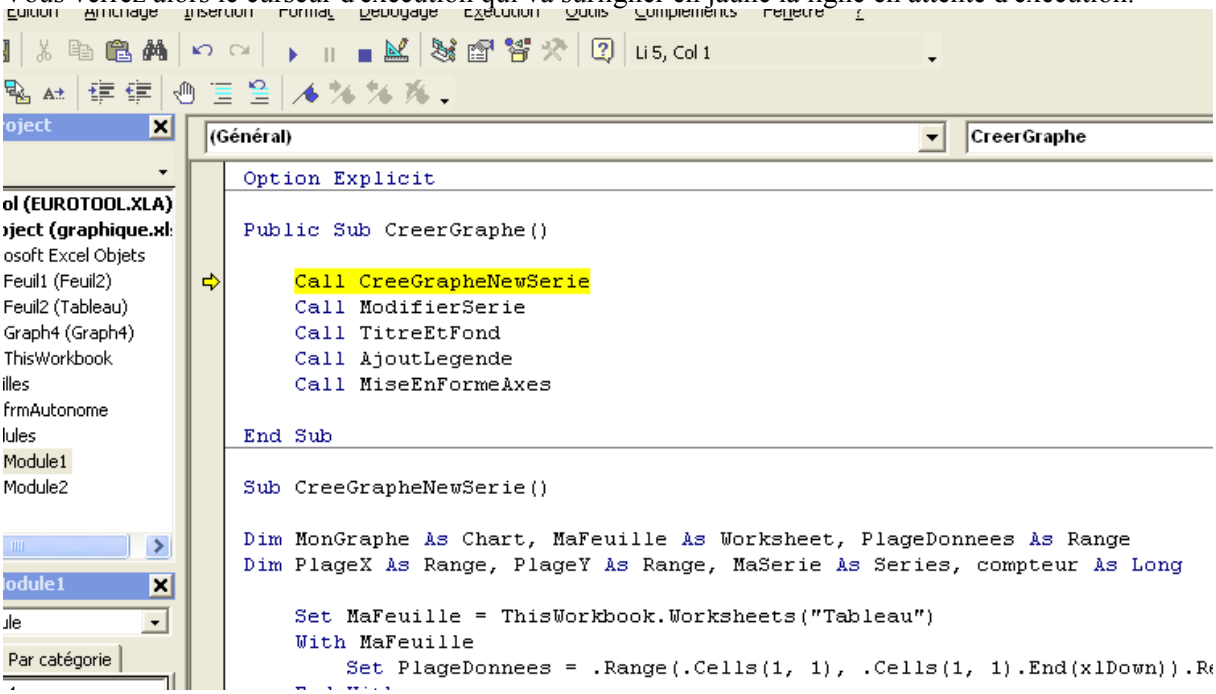
Avant de nous replonger dans du code pour la fin de ce charmant petit cours, retournons quelques instants dans l'environnement de développement à la découverte des outils de débogage. Pour comprendre plus facilement les erreurs d'exécution, l'environnement VBA va vous fournir quelques outils bien pratiques. Pour cela, nous continuerons avec les codes que nous avons vu précédemment, qui seront appelés d'une procédure :

```
Public Sub CreerGraphe()

    Call CreeGrapheNewSerie
    Call ModifierSerie
    Call TitreEtFond
    Call AjoutLegende
    Call MiseEnFormeAxes

End Sub
```

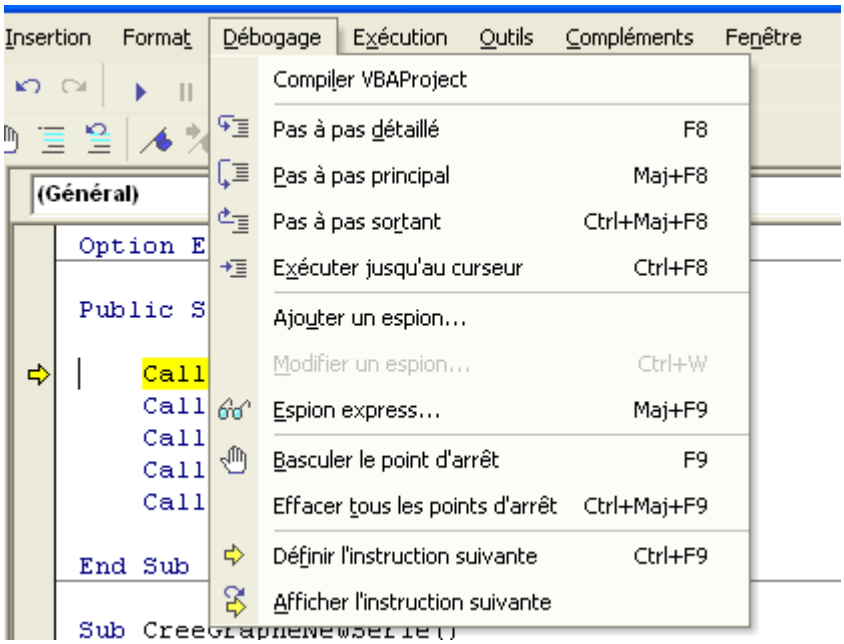
Exécution en mode pas à pas

Si vous voulez voir ce que fait votre code vous pouvez le faire exécuter en mode pas à pas c'est-à-dire en avançant le curseur d'exécution ligne après ligne. Pour cela vous devez sélectionner le menu "Débogage – Pas à pas détaillé" ou appuyer sur F8.

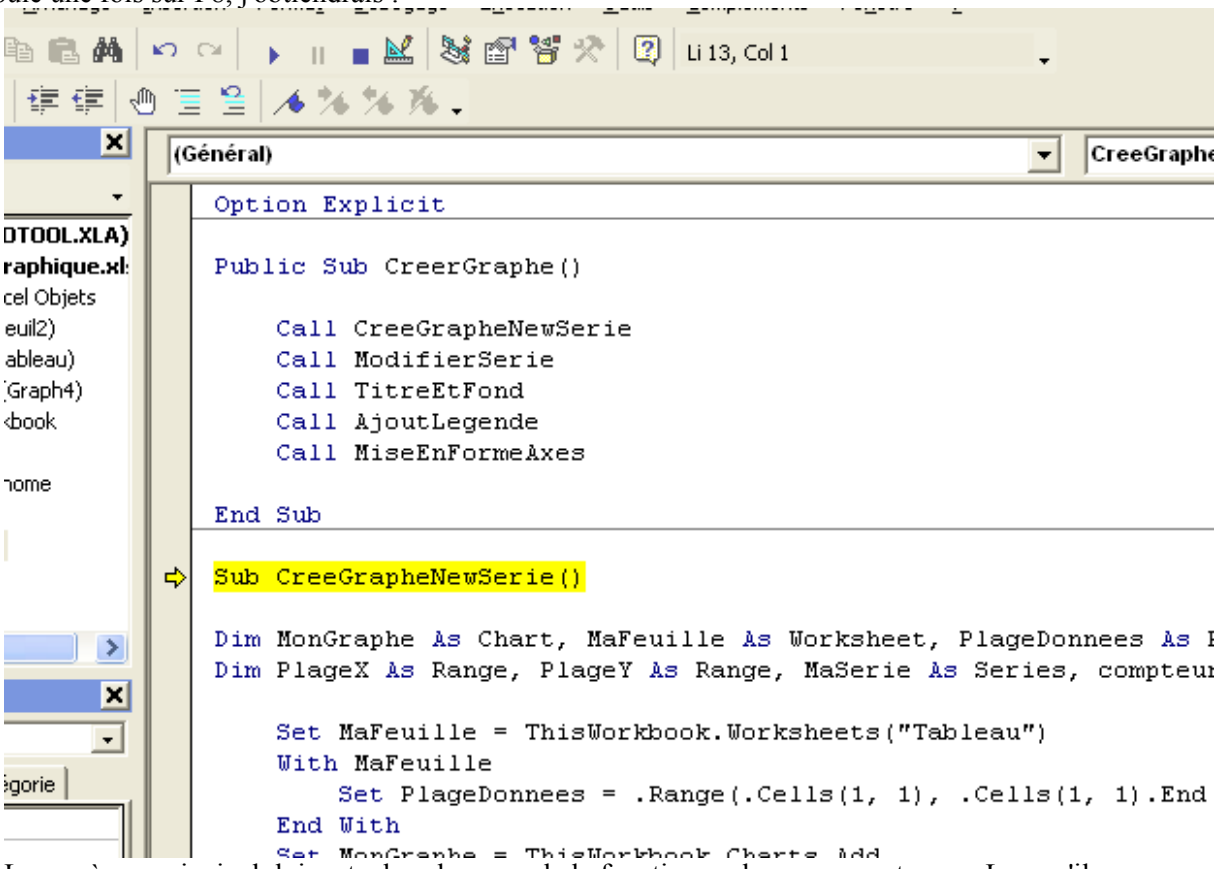
Vous verrez alors le curseur d'exécution qui va surligner en jaune la ligne en attente d'exécution.



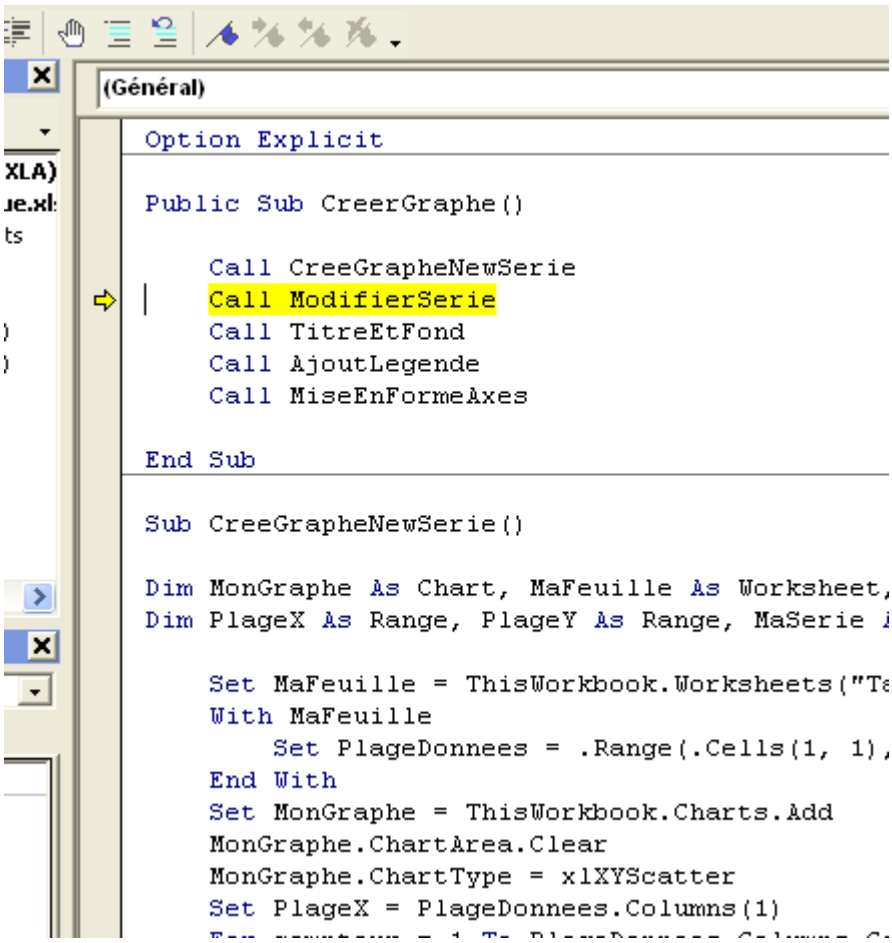
A chaque pression successive sur la touche F8, le curseur va avancer d'une ligne. Si vous regardez dans le menu débogage, vous voyez :



Le pas à pas détaillé suit exactement le curseur d'exécution, c'est-à-dire que lorsque le code appelle une fonction, le curseur va dans cette fonction. Ainsi dans la capture d'écran précédente, si j'appuie une fois sur F8, j'obtiendrais :



Le pas à pas principal, lui reste dans le corps de la fonction ou le curseur se trouve. Lorsqu'il passe sur une fonction, celle-ci est exécutée et le curseur va sur la ligne suivante de la même procédure. Si j'avais appuyé sur Maj+F8, j'aurais obtenu :



La petite flèche jaune que vous voyez sur la gauche de la fenêtre vous permet de déplacer le curseur avec la souris. Si le contexte le permet, vous pouvez donc vous placer sur une ligne particulière et appuyer sur F8 pour l'exécuter. La barre grise dans laquelle se trouve la flèche jaune s'appelle barre de déboguage. Dans certains cas, vous pouvez modifier les lignes de codes sans arrêter l'exécution.

Les points d'arrêts

Si votre code est un peu long ou s'il contient des boucles, cela peut être vite fastidieux d'appuyer sur F8 jusqu'à atteindre le point du code qui vous intéresse. Pour lancer l'exécution jusqu'à un point donnée vous pouvez ou placer le curseur sur la ligne et utiliser "Déboguage – Exécuter jusqu'au curseur" ou utiliser des points d'arrêts.

Pour placer un point d'arrêt, il suffit de cliquer dans la barre de déboguage au niveau de la ligne où vous souhaitez bloquer l'exécution;

```
Option Explicit

Public Sub CreerGraphe()

    Call CreeGrapheNewSerie
    Call ModifierSerie
    Call TitreEtFond
    Call AjoutLegende
    Call MiseEnFormeAxes

End Sub

Sub CreeGrapheNewSerie()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Range
Dim PlageX As Range, PlageY As Range, MaSerie As Series, compteur As Long

    Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
    With MaFeuille
        Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1, 1).End(xlDown)).Resiz
    End With
    Set MonGraphe = ThisWorkbook.Charts.Add
    MonGraphe.ChartArea.Clear
    MonGraphe.ChartType = xlXYScatter
    Set PlageX = PlageDonnees.Columns(1)
    For compteur = 1 To PlageDonnees.Columns.Count - 1
        Set PlageY = PlageX.Offset(, compteur)
        Set MaSerie = MonGraphe.SeriesCollection.NewSeries
        With MaSerie
            .Values = PlageY
            .XValues = PlageX
```

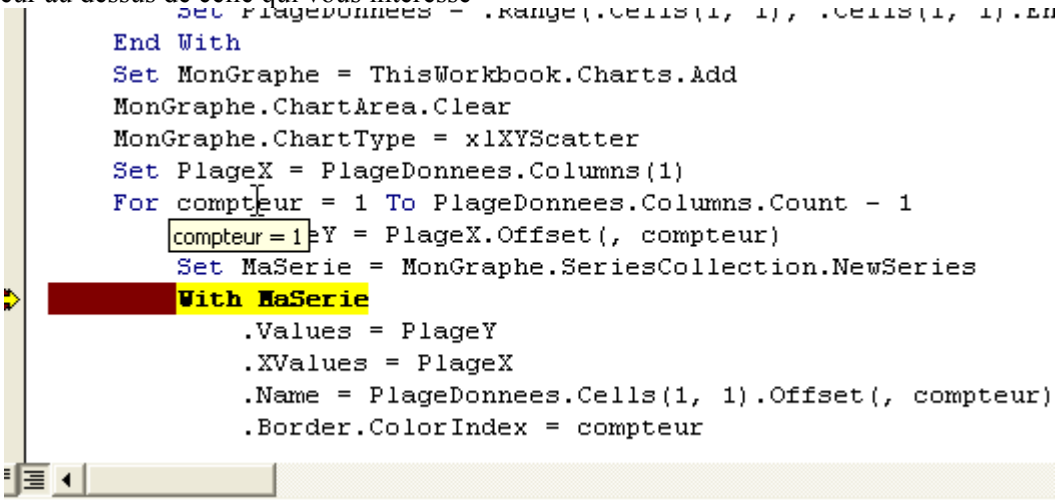
Choisissez ensuite exécuter dans le menu Exécution (F5), le code s'exécutera jusqu'au point d'arrêt.

```
Option Explicit

Public Sub CreerGraphe()

    Call CreeGrapheNewSerie
    Call ModifierSerie
    Call TitreEtFond
    Call AjoutLegende
    Call MiseEnFormeAxes

End Sub

Sub CreeGrapheNewSerie()

Dim MonGraphe As Chart, MaFeuille As Worksheet, PlageDonnees As Ra
Dim PlageX As Range, PlageY As Range, MaSerie As Series, compteur

    Set MaFeuille = ThisWorkbook.Worksheets("Tableau")
    With MaFeuille
        Set PlageDonnees = .Range(.Cells(1, 1), .Cells(1, 1).End(>
    End With
    Set MonGraphe = ThisWorkbook.Charts.Add
    MonGraphe.ChartArea.Clear
    MonGraphe.ChartType = xlXYScatter
    Set PlageX = PlageDonnees.Columns(1)
    For compteur = 1 To PlageDonnees.Columns.Count - 1
        Set PlageY = PlageX.Offset(, compteur)
        Set MaSerie = MonGraphe.SeriesCollection.NewSeries
        With MaSerie
            .Values = PlageY
            .XValues = PlageX
            .Name = PlageDonnees.Cells(1, 1).Offset(, compteur)
```

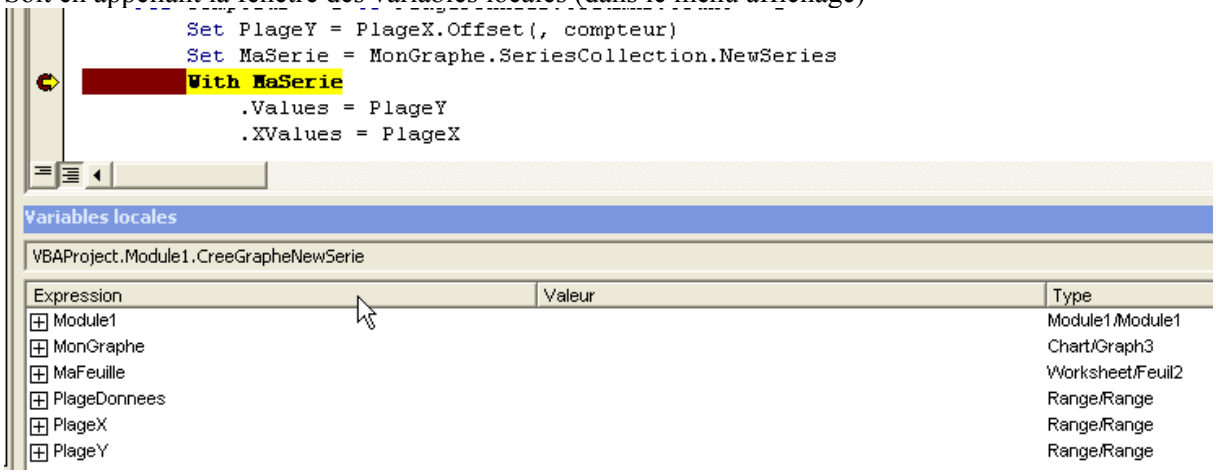
Vous pouvez placer plusieurs points d'arrêts dans votre code et les effacer tous en appuyant sur Ctrl+Maj+F9.

Variables locales

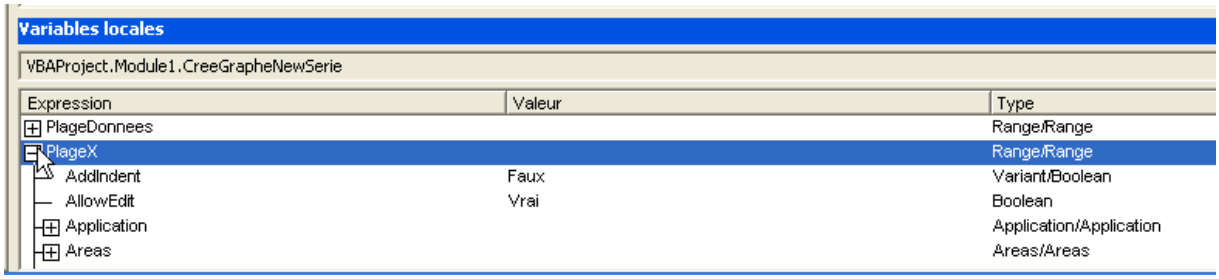
Vous pouvez voir au fur et à mesure du pas à pas la valeur de vos variables, soit en plaçant le curseur au dessus de celle qui vous intéresse



Soit en appellant la fenêtre des variables locales (dans le menu affichage)

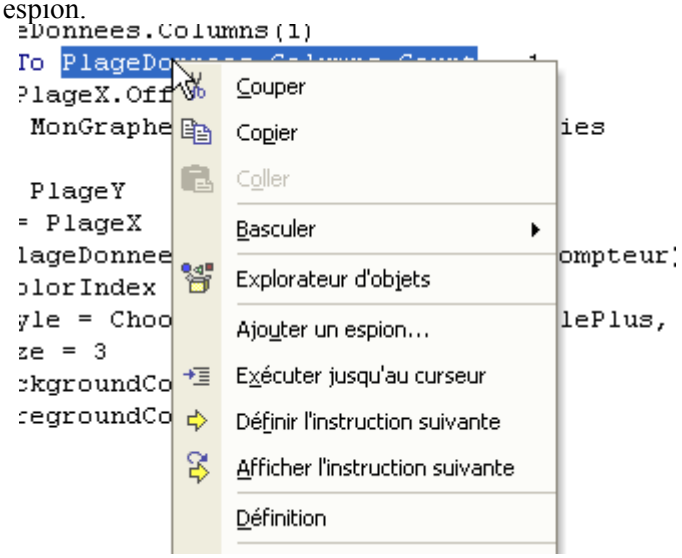


Lorsqu'il s'agit de variables objets, elles ont un signe + à coté d'elles lorsque l'objet est instancié. En cliquant dessus, vous verrez apparaître la liste des propriétés de l'objet et leurs valeurs à cet instant de l'exécution :

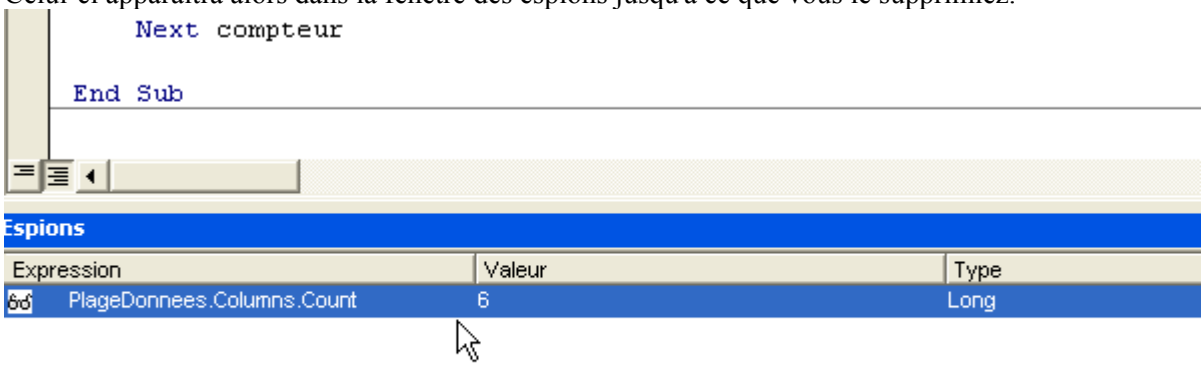


Les espions

Vous pouvez aussi choisir un élément en le surlignant, faire un clic droit et sélectionner Ajouter un espion.



Celui-ci apparaîtra alors dans la fenêtre des espions jusqu'à ce que vous le supprimiez.

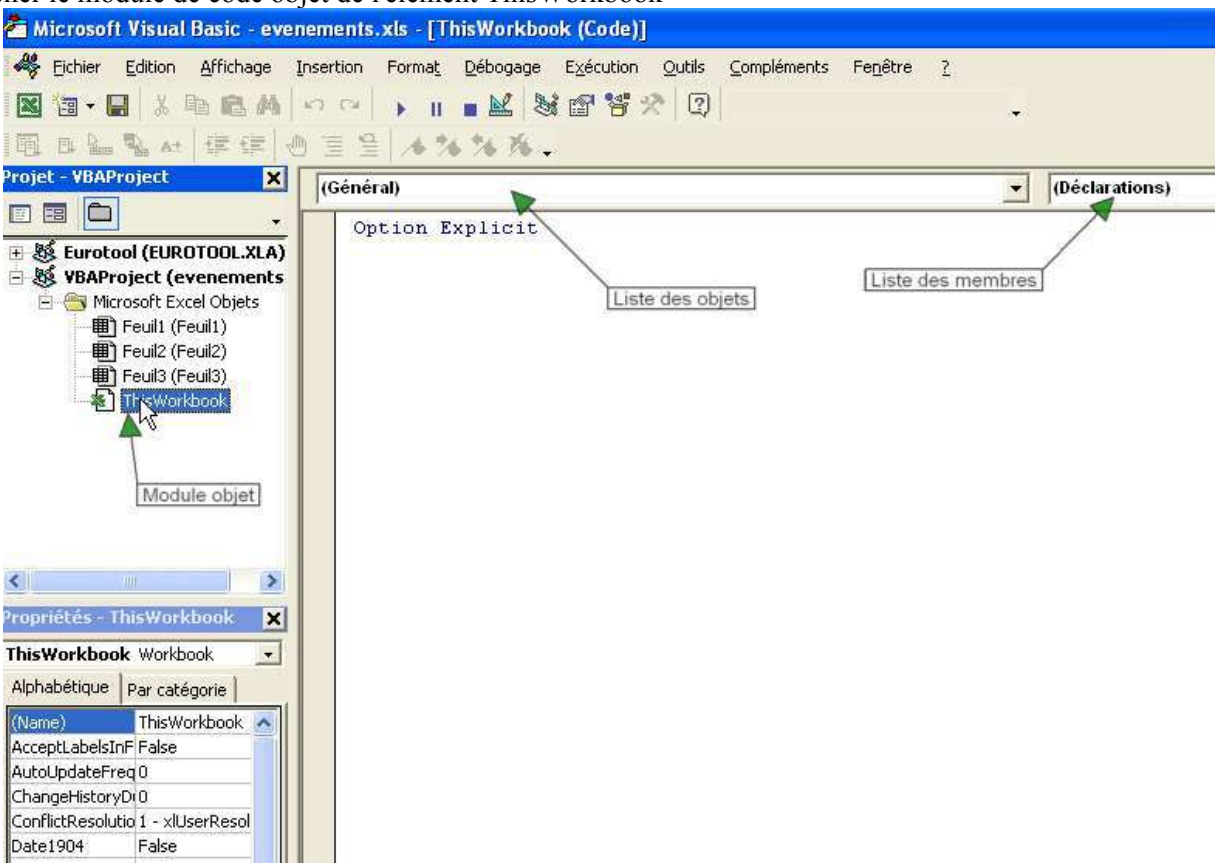


Manipuler les évènements

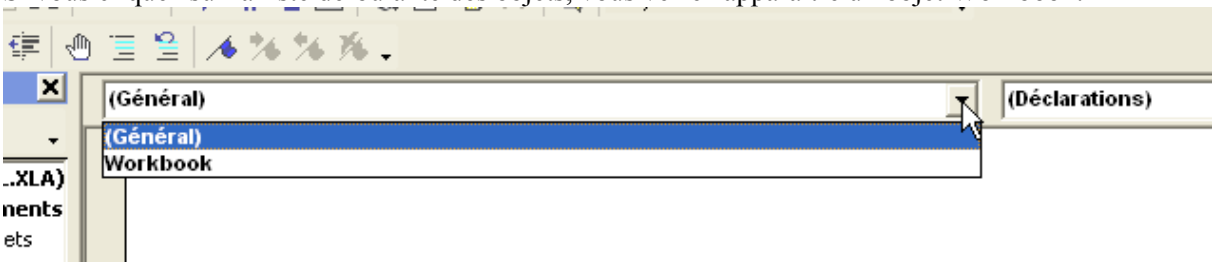
Nous allons maintenant aller un peu plus loin en manipulant quelques évènements des objets Excel que nous avons vu.

Un évènement, c'est une procédure qui s'exécute chaque fois qu'un évènement particulier se produit. Seuls les modules objets peuvent contenir des procédures évènementielles. Dans la partie du modèle objet que nous avons vu, seul les objets WorkBook, WorkSheet et Chart gèrent des évènements. L'objet Application en gère aussi mais nous ne les verrons pas dans ce cours.

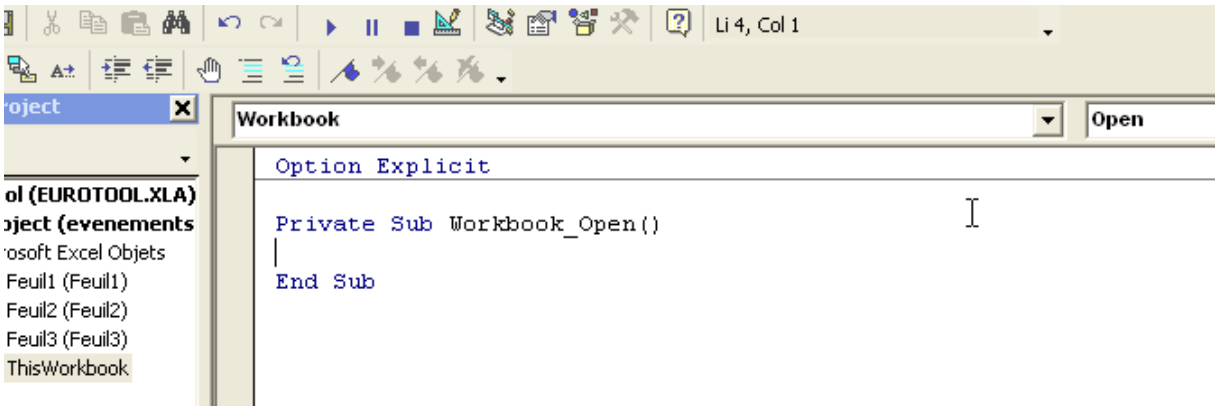
Pour gérer un évènement du classeur, double cliquons dans l'explorateur de projets ce qui va afficher le module de code objet de l'élément ThisWorkbook



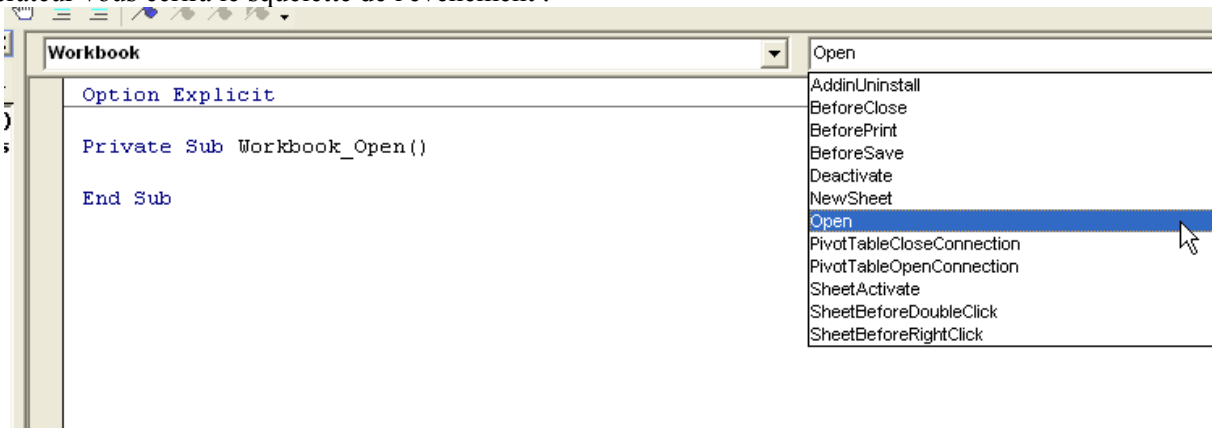
Si vous cliquez sur la liste déroulante des objets, vous verrez apparaître un objet Workbook.

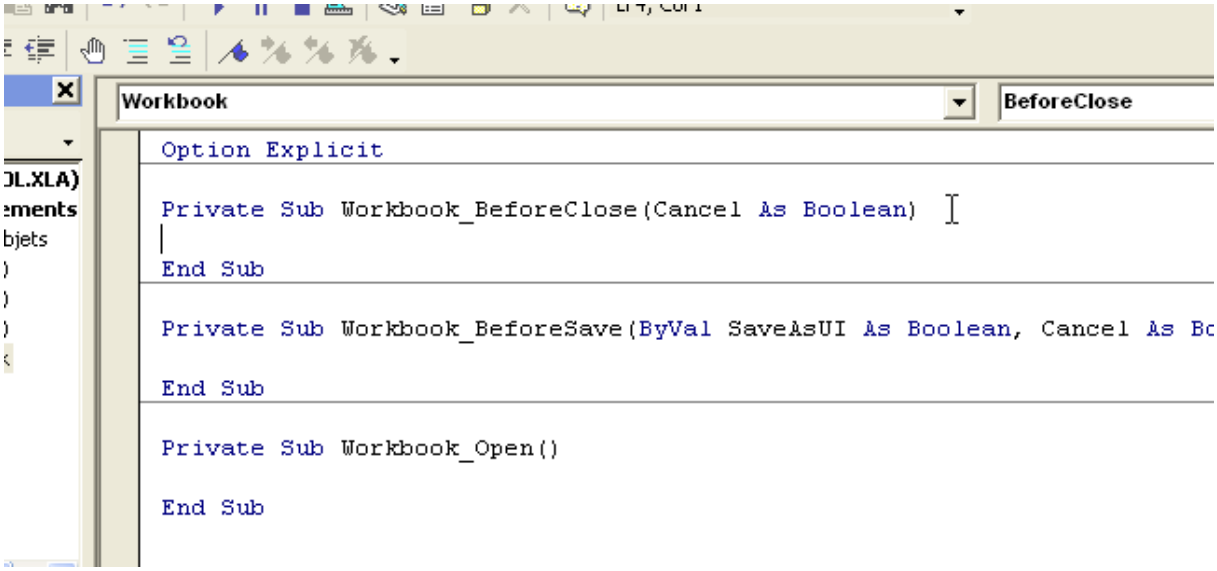


Sélectionner le, vous allez alors voir apparaître le code :



Notez que la valeur Open apparaît dans la liste des membres. Si vous cliquez sur celle-ci vous voyez apparaître la liste des évènements disponible pour le classeur. Sélectionnez un membre et le générateur vous écrira le squelette de l'évènement :





Comme vous le voyez, certains évènements ont des arguments prédéfinis, d'autres non. La définition de l'évènement ne peut être changée.

Prenons donc quelques exemples pour l'objet Workbook.

Le code suivant masque la feuille tableau si le classeur est ouvert en lecture seule

```
Private Sub Workbook_Open()
    If Me.ReadOnly Then
        Me.Worksheets("Tableau").Visible = xlSheetVeryHidden
    End If
End Sub
```

Lorsqu'il y a des arguments, vous pouvez les utiliser comme dans n'importe quelle procédure, à la condition de savoir à quoi ils servent. Par exemple, l'argument Cancel de l'évènement BeforeClose annule la fermeture du classeur. Le code suivant permet de forcer la saisie d'un identifiant avant sa fermeture.

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
Dim Reponse As String, Nom As Name

    Reponse = Application.InputBox("Entrez votre identifiant", "Contrôle",
, , , , , 2)
    Cancel = True
    For Each Nom In Me.Names
        If StrComp(Mid(Nom.RefersTo, 2), Reponse, vbTextCompare) = 0 Then
            Cancel = False
            Exit For
        End If
    Next Nom
End Sub
```

Ce code est un gadget, on utilise plus généralement cet évènement pour restaurer l'objet application lorsqu'on l'a modifié.

Evènements de feuille de calcul

Nous allons voir ici quelques évènements gérés par la feuille de calcul. Sachez qu'il en existe d'autres, et que tous ces évènements remontent aussi vers des évènements gérés au niveau du classeur préfixé par le mot 'Sheet' et acceptant un argument 'sh' identifiant la feuille dans lequel l'évènement se produit. Autrement dit l'évènement Activate de la feuille 1 lèvera aussi l'évènement SheetActivate du classeur en passant un argument sh qui sera une référence à la feuille 1.

Activation de la feuille

Défini par

```
Private Sub Worksheet_Activate()
```

Se produit lorsque la feuille est activée, c'est-à-dire que la feuille est sélectionnée par le code ou par l'utilisateur. On l'utilise généralement lorsqu'on souhaite exécuter des actions (calculs, mise en forme, connexion à une source de données, etc.) sur la feuille uniquement lorsque celle-ci peut être accessible à l'utilisateur c'est-à-dire lorsqu'elle devient la feuille active.

Imaginons par exemple que nous avons une feuille appelée "Récapitulatif", qui contient un grand nombre de calculs issus de données situées dans d'autres feuilles du classeur. Nous avons bloqué le mode de calcul du classeur pour que le changement des données n'engendre pas à chaque fois un recalcul assez long. Lorsque l'utilisateur va aller voir son récapitulatif, il va bien falloir forcer le calcul pour que les éléments visualisés reflètent bien les valeurs actualisées. Nous écrirons donc dans le module objet de la feuille.

```
Private Sub Worksheet_Activate()
    Me.Calculate
End Sub
```

Cependant se code peut poser un problème de comportement, car la feuille active par défaut, c'est-à-dire celle qui s'affiche lors de l'ouverture du classeur ne lève pas l'évènement Activate. Bien qu'en toute rigueur, cela ne pose pas de problème sauf en cas de liaison externe, il convient généralement de s'assurer que la feuille contenant cet évènement ne soit pas la feuille active à l'ouverture, et donc qu'elle ne soit pas la feuille active lors de l'enregistrement du classeur. Pour cela on utilise l'évènement BeforeSave du classeur tel que :

```
Private Sub Workbook_BeforeSave(ByVal SaveAsUI As Boolean, Cancel As
Boolean)
    If Me.ActiveSheet Is Me.Worksheets("Recapitulatif") Then
        Me.Worksheets("Donnees").Activate
    End If
End Sub
```

Gestion du clic droit

L'évènement BeforeRightClick se déclenche lorsque l'utilisateur fait un clic droit sur une cellule ou sur une sélection de cellule **avant** que le menu contextuel ne s'affiche. La définition est :

```
Private Sub Worksheet_BeforeRightClick(ByVal Target As Range, Cancel As
Boolean)
```

Où Target est un objet Range qui désigne la cellule ou la sélection recevant l'évènement, et Cancel un booléen permettant d'annuler l'affichage du menu contextuel.

Généralement, on utilise cet évènement pour restreindre ou pour ajouter des éléments au menu contextuel. Prenons l'exemple suivant qui supprime l'accès à la fonction 'couper' du menu contextuel lorsqu'on clique sur une cellule de la plage "A1:E5".

```
Private Sub Worksheet_BeforeRightClick(ByVal Target As Range, Cancel As
Boolean)

    If Not Application.Intersect(Target, Me.Range("A1:E5")) Is Nothing Then
        Application.CommandBars("Cell").Controls(1).Visible = False
    Else
        Application.CommandBars("Cell").Controls(1).Visible = True
    End If

End Sub
```

Outre l'exemple direct, nous trouvons un exemple classique de l'utilisation de l'argument Target dans les évènements de feuille. Détaillons-en un peu le principe. Généralement l'évènement ne doit concerner que quelques cellules et non toutes les cellules de la feuille. Pour gérer cela, on crée une plage nommée pour les cellules qui doivent gérer l'évènement où on appelle directement l'objet Range comme dans l'exemple s'il s'agit d'une plage rectangulaire. On récupère l'intersection entre l'argument Target et la plage cible, si cette intersection vaut Nothing, c'est qu'aucune cellule de la plage devant lever l'évènement ne l'a reçu, sinon au moins une des cellules l'a reçu.

Notez aussi que l'objet Range est qualifié par le mot clé Me. En effet, le code de l'évènement est forcément contenu dans le module objet de la feuille. Le mot clé Me fait donc référence à la feuille ce qui permet de l'utiliser en qualificateur de la propriété Range.

Changement de sélection

L'évènement SelectionChange se produit lorsqu'au moins une nouvelle cellule est sélectionnée. Cet évènement suit la définition :

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

Cet évènement bien que très pratique peut avoir des effets de bords néfastes dans certains cas, si on déclenche une cascade d'évènements. Prenons l'exemple suivant :

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
    If Not Application.Intersect(Target, Me.Columns("A:J")) Is Nothing Then
        Target.Offset(1).Select
    End If
End Sub
```

Ce code va entraîner un déplacement rapide de la sélection vers le bas de la feuille. En effet, l'appel de la méthode Select va lever à nouveau un évènement SelectionChange qui va lui-même lever aussi un évènement SelectionChange et ainsi de suite, c'est la cascade d'évènement. Généralement on peut contourner le problème en travaillant sur la gestion de l'argument Target, mais parfois on ne peut pas. Dans ce cas on désactive temporairement la gestion des évènements d'Excel en jouant sur la propriété EnableEvents de l'objet Application, tel que :

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
    Application.EnableEvents = False
    If Not Application.Intersect(Target, Me.Columns("A:J")) Is Nothing Then
        Target.Offset(1).Select
    End If
    Application.EnableEvents = True
End Sub
```

Changement de valeur

L'évènement Change, souvent confondu avec l'évènement précédent, se produit lorsque la valeur d'une cellule change. Suit la définition :

```
Private Sub Worksheet_Change(ByVal Target As Range)
```

L'évènement est levé après le changement de la valeur, il n'est pas possible de restaurer simplement la valeur précédente de la cellule. Tout comme le précédent, cet évènement est susceptible de déclencher assez aisément des évènements en cascade, on évite donc généralement d'affecter une valeur aux cellules de l'argument Target au sein de cet évènement.

Manipuler les contrôles

Pour finir ce cours déjà assez long, nous allons traiter assez succinctement de la manipulation des contrôles et des formulaires. On appelle contrôle des éléments graphiques ayant leur propre code de fonctionnement qui s'intègre aux éléments Excel. Il existe plusieurs milliers de contrôles intégrables dans Excel, puisque tous les contrôles ActiveX le sont, mais nous ne traiterons que des contrôles proposés par défaut par Excel.

Deux familles de contrôles

Historiquement, Excel possédait un jeu de contrôles spécifiques permettant de faire des formulaires simplifiés. Pour des raisons de compatibilité ascendante, ces contrôles sont toujours disponibles, même si une partie de leur documentation n'est plus accessible. Microsoft recommande de privilégier l'utilisation des contrôles ActiveX, cependant les contrôles formulaires restent bien adaptés et assez simple d'utilisation dans quelques cas.

Les contrôles formulaires

Si vous n'avez jamais utilisé les contrôles formulaires, vous pouvez sauter cette partie.

Les contrôles formulaires sont accessibles au travers de la collection Shapes. Certains comme les zones d'édition ou les zones combinées ont été rendus inaccessibles pour les nouveaux classeurs. Cependant il reste quatre contrôles encore utilisés de temps en temps.

- Le contrôle bouton.
- La case à cocher
- La case d'option
- La zone combinée déroulante

Avantages

Il faut bien reconnaître qu'ils n'en ont pas beaucoup. Il n'y a rien qu'on puisse faire avec ces contrôles qu'on ne puisse pas obtenir avec leurs homologues Ms-Forms si ce n'est une certaine simplicité de liaison avec les données de la feuille.

En effet, leur logique d'utilisation est un peu différente de celle des contrôles ActiveX. Comme il s'agit de contrôles 'purement' Excel, ils trouvent leurs données de fonctionnement dans Excel obligatoirement et généralement dans le classeur qui les contient.

Le contrôle bouton lui garde un avantage certain, il est possible de lui affecter n'importe quelle procédure publique comme évènement Clic. De fait, il n'est pas obligatoire de gérer un code évènementiel spécifique, lui attribuer une procédure comme propriété OnAction suffit.

Ces contrôles sont encore utilisés parce que quand on les connaît bien, ils sont assez simples à manipuler.

Inconvénients

Ils sont nettement moins faciles (encore que) à manipuler par le code et assez peu configurables. A part cela il n'y a pas grand-chose à leur reprocher.

Exemples

Je vous disais plus haut qu'ils sont assez difficiles à manipuler. Comme toujours, tout dépend de comment on s'y prend. Sur bien des aspects, ils sont aussi simples à gérer que les contrôles ActiveX.

La difficulté apparente vient de la méthode à employer pour obtenir une référence sur le contrôle à manipuler et de trouver ses propriétés spécifiques.

L'erreur vient souvent du fait qu'on tente d'accéder à l'objet au travers de la collection Shapes. En effet, ils sont de type générique Shape.

On y accède donc en écrivant

```
Feuille.Shapes(Nom contrôle")
```

Seulement cela renvoie un objet Shape qui ne présente pas aisément les propriétés intéressantes de nos contrôles. En fait, les contrôles formulaires incorporés sont exposés à travers plusieurs collections spécifiques de l'objet Worksheet. Dans le cas qui nous intéresse, il s'agit des collections, Buttons, CheckBoxes, OptionButtons et DropDowns.

Elles se manipulent comme toutes les collections ce qui simplifie énormément la programmation.

Le code suivant va mettre en place un formulaire.

```
Public Sub Calculer()
    ThisWorkbook.Worksheets("Recapitulatif").Calculate
End Sub

Public Sub ConstructForm()
    'ajoute un bouton de calcul
    Dim Feuille As Worksheet, rngPos As Range, rngVal As Range
    Dim zlist As DropDown, compteur As Long, Opt As OptionButton

    Set Feuille = ThisWorkbook.Worksheets("Recapitulatif")
    Set rngPos = Feuille.Cells(1, 1)
    'ajout du bouton
    With Feuille.Buttons.Add(rngPos.Left, rngPos.Top, rngPos.Resize(,
2).Width, rngPos.Height)
        .Caption = "Calculer maintenant"
        .OnAction = "Calculer"
    End With
    'ajout d'une case à cocher
    Set rngPos = rngPos.Offset(2)
    Set rngVal = Feuille.Range("F1")
    Feuille.CheckBoxes.Add rngPos.Left, rngPos.Top, rngPos.Width,
rngPos.Height
    With Feuille.CheckBoxes(1)
        .Caption = "Jours feriés inclus"
        .Display3DShading = True
        .LinkedCell = rngVal.Address(True, True, xlA1)
        .Value = False
    End With
    'ajout de sept bouton d'option
    Set rngPos = rngPos.Offset(-2, 3)
    Set rngVal = rngVal.Offset(2)
    For compteur = 1 To 7
        Set Opt = Feuille.OptionButtons.Add(rngPos.Left, rngPos.Top,
rngPos.Width, rngPos.Height)
        Opt.Caption = WeekdayName(compteur, False, vbMonday)
        Opt.LinkedCell = rngVal.Offset(compteur - 1).Address(True, True,
xlA1)
        Opt.Value = False
        Set rngPos = rngPos.Offset(1)
    Next compteur
    'ajout d'une zone de liste
    Set rngPos = Feuille.Range("A5")
    Feuille.Range("G1").Value = "Janvier"
    Feuille.Range("G2").Value = "Février"
    Feuille.Range("G1:G2").AutoFill Destination:=Feuille.Range("G1:G12"),
Type:=xlFillDefault
    Set zlist = Feuille.DropDowns.Add(rngPos.Left, rngPos.Top,
rngPos.Resize(, 2).Width, rngPos.Height)
    With zlist
        .DropDownLines = 6
```

```
        .LinkedCell = rngVal.Offset(10)
        .ListFillRange = "G1:G12"
    End With

End Sub
```

Les contrôles MsForms

Ce sont les contrôles que l'on utilise pour les formulaires VBA. Ces contrôles sont de la famille des contrôles ActiveX, c'est-à-dire des contrôles fournis par un composant. Le composant s'appelle MsForms, il est accessible indifféremment pour les feuilles ou pour les formulaires.

Quel que soit leur conteneur (Feuille de calcul ou UserForm), ils peuvent être ajoutés à la création (Design Time) parfois appelé ajout statique, ou à l'exécution (Run Time) qui est l'ajout dynamique. Pour créer dynamiquement un contrôle, il faut de toute façon connaître le nom de sa classe.

Nous ne trouverons dans ce tableau que les contrôles fournis par le composant MsForms.

Contrôle	Classe
Bouton	Forms.CommandButton.1
Zone de texte	Forms.TextBox.1
Case à cocher	Forms.CheckBox.1
Case d'option	Forms.OptionButton.1
Liste	Forms.ListBox.1
Liste déroulante	Forms.ComboBox.1
Bouton bascule	Forms.ToggleButton.1
UpDown	Forms.SpinButton.1
Barre de défilement	Forms.ScrollBar.1

La manipulation des contrôles est ensuite sensiblement identique, puisque ce sont de objets, on agit donc sur les propriétés et méthodes des contrôles et on gère leurs évènements.

Contrôles incorporés

Les contrôles incorporés sont des contrôles placés sur une feuille de calcul. Ils appartiennent à la collection OLEObjects de l'objet Worksheet. Il n'existe pas de collection spécifique par type de contrôle similaire à ce que nous avons vu précédemment, vous devez donc passer par la collection OLEObjects pour créer ou pour atteindre un contrôle existant.

Commençons par recréer le même formulaire que dans l'exemple précédent.

```
Public Sub ConstructFormActiveX()
    'ajoute un bouton de calcul
    Dim Feuille As Worksheet, rngPos As Range, rngVal As Range
    Dim zlist As MSForms.ComboBox, compteur As Long, Opt As
MSForms.OptionButton

    Set Feuille = ThisWorkbook.Worksheets("Recapitulatif")
    Set rngPos = Feuille.Cells(1, 1)
    'ajout du bouton
    With Feuille.OLEObjects.Add(ClassType:="Forms.CommandButton.1",
Link:=False _
        , DisplayAsIcon:=False, Left:=rngPos.Left, Top:=rngPos.Top,
Width:=rngPos.Resize(, 2).Width _
        , Height:=rngPos.Height).Object
        .AutoSize = True
        .Caption = "Calculer maintenant"
    End With
    'ajout d'une case à cocher
    Set rngPos = rngPos.Offset(2)
    Set rngVal = Feuille.Range("F1")
    Feuille.OLEObjects.Add "Forms.CheckBox.1", , , , , , , rngPos.Left,
rngPos.Top, rngPos.Width, rngPos.Height
    With Feuille.OLEObjects(Feuille.OLEObjects.Count).Object
```

```
        .AutoSize = True
        .Caption = "Jours feriés inclus"
        .Value = False
    End With
    'ajout de sept bouton d'option
    Set rngPos = rngPos.Offset(-2, 3)
    Set rngVal = rngVal.Offset(2)
    For compteur = 1 To 7
        rngPos.EntireRow.RowHeight = 15
        Set Opt = Feuille.OLEObjects.Add("Forms.OptionButton.1", , , , , ,
, rngPos.Left, rngPos.Top, rngPos.Width, rngPos.Height).Object
        Opt.Caption = WeekdayName(compteur, False, vbMonday)
        Opt.Value = False
        Set rngPos = rngPos.Offset(1)
    Next compteur
    'ajout d'une zone de liste
    Set rngPos = Feuille.Range("A5")
    Set zlist = Feuille.OLEObjects.Add("Forms.ComboBox.1", , , , , , ,
rngPos.Left, rngPos.Top, rngPos.Resize(, 2).Width, rngPos.Height).Object
    With zlist
        .ListRows = 6
        For compteur = 1 To 12
            zlist.AddItem MonthName(compteur)
        Next compteur
    End With

End Sub
```

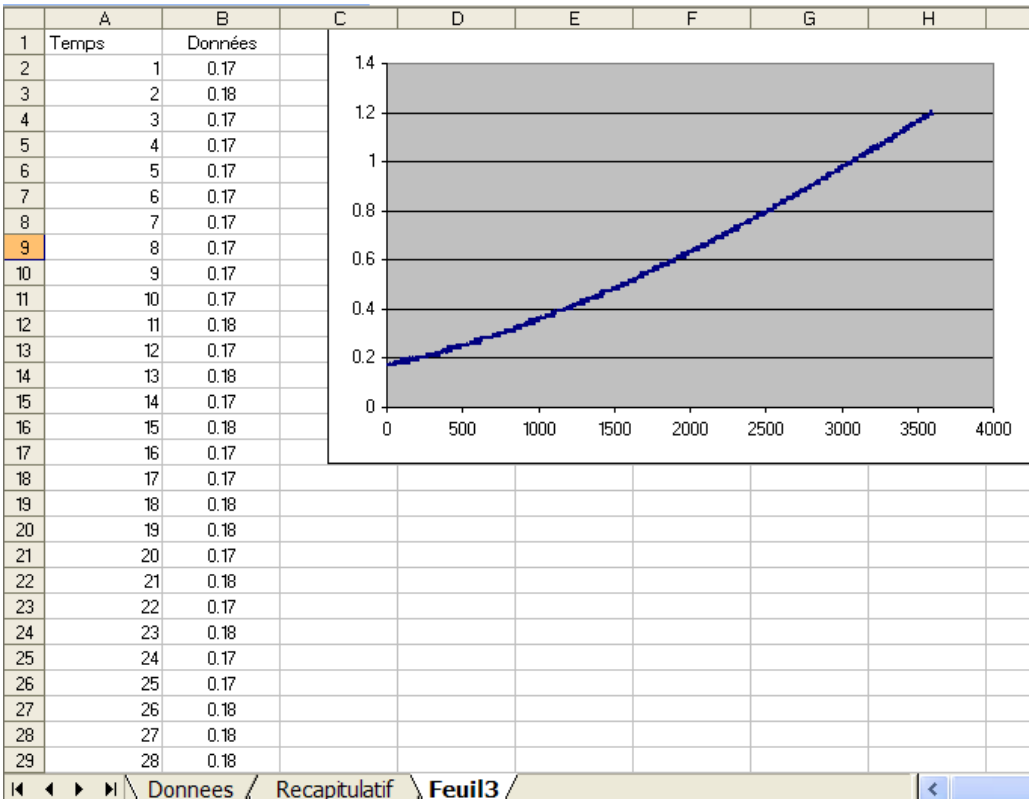
Comme vous le voyez, l'ensemble des contrôles est ajouté par le biais de la méthode Add de la collection OLEObjects. Vous noterez également que les contrôles n'utilisent plus de données liées sur la feuille de calcul. Non qu'ils ne sauraient pas le faire, mais parce qu'ils n'en ont pas nécessairement besoin. En effet, les contrôles ActiveX gèrent leurs propriétés en interne, celles-ci sont sauvegardées lorsqu'on sauvegarde le classeur, il n'y a donc plus nécessité de les stocker dans des cellules du classeur.

Si l'approche reste assez similaire, vous noterez que pour la gestion du bouton de commande, le cas est nettement plus complexe. En effet, pour qu'un bouton ActiveX déclenche une action lorsqu'on clique dessus, il faut gérer son évènement clic. Or sur un contrôle ajouté dynamiquement, il n'est pas aisé d'ajouter dynamiquement le code de l'évènement Clic. Ceci voudrait dire qu'on devrait écrire dans le module de la feuille qui va contenir le bouton la procédure évènementielle Clic avant que le bouton n'existe. Ceci n'a pas vraiment d'intérêt. On ne génère donc que très rarement des contrôles boutons dynamiquement.

Pour gérer un évènement de contrôle ActiveX incorporés dans Excel, il faut écrire le code d'évènement dans le module de la feuille qui contient le contrôle, ce qui dans notre cas donnerait un code du type :

```
Private Sub CommandButton1_Click()

    Call Calculer

End Sub
```

Il est possible de réaliser des opérations relativement complexes avec des codes assez simples. Imaginons le scénario suivant. Nous avons une série de donnée pseudo linéaire, que nous pourrions représenter sous la forme suivante :



Il faudrait pouvoir créer un système permettant de définir simplement une limite haute et une limite basse d'une zone de la courbe sur laquelle nous voudrions calculer une régression linéaire. Dit comme cela, ça à l'air compliqué et pourtant c'est relativement simple à faire.

Pour que les limites apparaissent sur le graphique, nous devons générer deux pseudos séries créant des traits verticaux en début et en fin de courbe. Pour ne pas gêner l'utilisateur, nous écrirons les données de ses séries dans des cellules situées sous le graphique afin qu'elle ne soit pas visible, et nous les masquerons aussi dans la légende. Comme il n'existe pas de méthode simple pour faire glisser les limites en partant du graphique, nous allons utiliser deux contrôles barre de défilement (ScrollBar) qui simuleront ce déplacement. Enfin nous ajouterons le calcul de régression et un affichage de celui-ci dans le graphique.

```
Public Sub Regression()

    Dim Feuille As Worksheet, LimBasse As Range, LimHaute As Range
    Dim Graphique As Chart, MaSerie As Series, Barre As MSForms.ScrollBar
    Dim Graphe As ChartObject, Affichage As String

    Set Feuille = ThisWorkbook.Worksheets("Feuil3")
    Set Graphe = Feuille.ChartObjects(1)
    Set Graphique = Graphe.Chart
    'crée deux plages de deux couples de valeur pour les limites
    Set LimBasse = Feuille.Range("E1:F2")
    Set LimHaute = Feuille.Range("E4:F5")
    'récupére les valeurs max des axes pour initialiser les limites
    LimBasse.Cells(1).Value = Graphique.Axes(xlCategory).MinimumScale
    LimBasse.Cells(2).Value = Graphique.Axes(xlValue).MinimumScale
    LimBasse.Cells(3).Formula = "=" & LimBasse.Cells(1).Address
    LimBasse.Cells(4).Value = Graphique.Axes(xlValue).MaximumScale
    LimHaute.Cells(1).Value = Graphique.Axes(xlCategory).MaximumScale
    LimHaute.Cells(2).Value = Graphique.Axes(xlValue).MinimumScale
    LimHaute.Cells(3).Formula = "=" & LimHaute.Cells(1).Address
    LimHaute.Cells(4).Value = Graphique.Axes(xlValue).MaximumScale
    'ajoute les limites au graphe
    Set MaSerie = Graphique.SeriesCollection.NewSeries
    With MaSerie
```

```
        .Values = LimBasse.Cells(2).Resize(2)
        .XValues = LimBasse.Cells(1).Resize(2)
        .MarkerStyle = xlMarkerStyleNone
        With .Border
            .LineStyle = xlContinuous
            .Weight = xlMedium
        End With
    End With
    Set MaSerie = Graphique.SeriesCollection.NewSeries
    With MaSerie
        .Values = LimHaute.Cells(2).Resize(2)
        .XValues = LimHaute.Cells(1).Resize(2)
        .MarkerStyle = xlMarkerStyleNone
        With .Border
            .LineStyle = xlContinuous
            .Weight = xlMedium
        End With
    End With
    'ajout du premier controle scrollbar
    Set Barre = Feuille.OLEObjects.Add(ClassType:="Forms.ScrollBar.1",
Left:=Graphe.Left, Top:=Graphe.BottomRightCell.Offset(2).Top,
Width:=Graphe.Width,
Height:=Graphe.BottomRightCell.Offset(2).Height).Object
    With Barre
        .BackColor = vbBlue
        .Min = LimBasse.Cells(1).Value
        .Max = LimHaute.Cells(1).Value
        .LinkedCell = LimBasse.Cells(1).Address(True, True, xlA1)
    End With
    'ajout du second controle scrollbar
    Set Barre = Feuille.OLEObjects.Add(ClassType:="Forms.ScrollBar.1",
Left:=Graphe.Left, Top:=Graphe.BottomRightCell.Offset(4).Top,
Width:=Graphe.Width,
Height:=Graphe.BottomRightCell.Offset(4).Height).Object
    With Barre
        .BackColor = vbGreen
        .Min = LimBasse.Cells(1).Value
        .Max = LimHaute.Cells(1).Value
        .LinkedCell = LimHaute.Cells(1).Address(True, True, xlA1)
    End With
    'gestion de la regression
    With Feuille.Range("G1")
        .FormulaR1C1Local = "=INDEX(L2C1:L3601C2;L1C5;1)"
        .Offset(1).FormulaR1C1Local =
"=INDEX(DROITEREG(DECALER(L1C2;L1C5;;L4C5-L1C5);DECALER(L1C1;L1C5;;L4C5-
L1C5);VRAI;VRAI);1)"
        .Offset(2).FormulaR1C1Local =
"=INDEX(DROITEREG(DECALER(L1C2;L1C5;;L4C5-L1C5);DECALER(L1C1;L1C5;;L4C5-
L1C5);VRAI;VRAI);2)"
        .Offset(3).FormulaR1C1Local = "=""y = "" & ARRONDI(L(-2)C;6) &""x +
"" & ARRONDI(L(-1)C;6)"
        Affichage = "=" & Feuille.Name & "!" & .Offset(3).Address(True,
True, xlA1)
        With Graphique.TextBoxes.Add(10, 10, 200, 50)
            .Formula = Affichage
        End With
    End With

End Sub
```

Il s'agit d'un code assez simple pour gérer une opération qui elle est complexe. La complexité de cet exercice réside plus dans l'écriture des formules permettant d'obtenir la régression que dans le code en lui-même.

UserForm

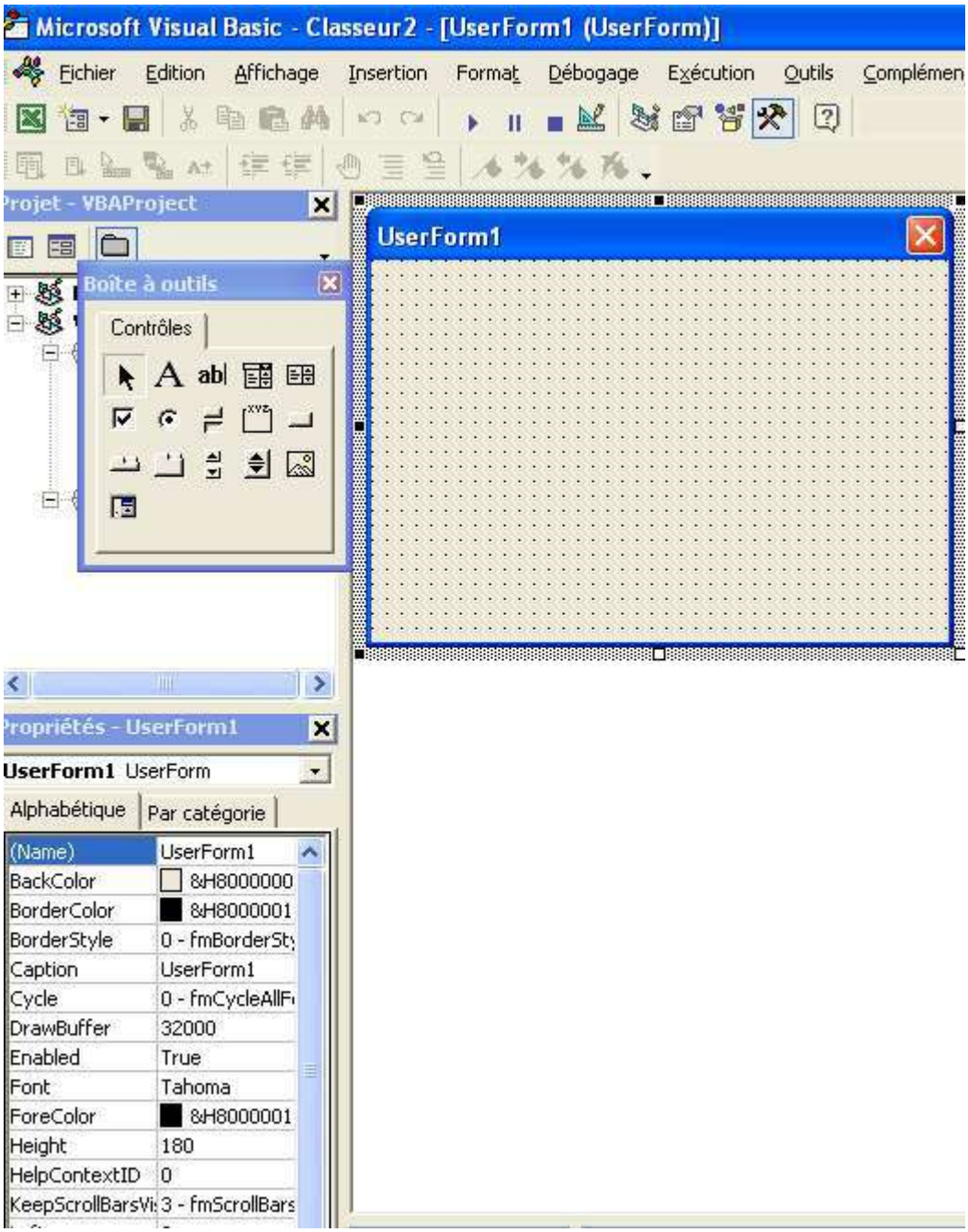
Nous n'allons ici que survoler la programmation des formulaires. Il s'agit d'un sujet extrêmement vaste que nous ne traiterons pas en détail dans ce cours.

Parfois appelées boîtes de dialogue, formulaire, feuille…le UserForm sert à générer une feuille contenant des contrôles permettant généralement d'interagir avec l'utilisateur.

Commençons donc par créer un UserForm. Ouvrez un classeur contenant des séries de données que l'on souhaite tracer, dans notre cas :

	A	B	C	D	E	F
1	Temps (s)	T° 1	T° 2	T° 3	T° 4	Vitesse
2	0	40.6	21.5	37.4	40.7	2497
3	1	40.7	21.5	37.5	40.8	2500
4	2	41.2	21.5	37.6	41	2500
5	3	41.1	21.5	37.8	41.2	2500
6	4	41.5	21.5	37.9	41.3	2495
7	5	41.2	21.5	38.1	41.4	2500
8	6	41.8	21.5	38.2	41.6	2497
9	7	41.5	21.5	38.4	41.8	2492
10	8	42.3	21.5	38.4	41.9	2495
11	9	42.4	21.5	38.6	42.1	2495
12	10	42.5	21.5	38.8	42.2	2497
13	11	43	21.5	38.9	42.3	2500
14	12	43.2	21.5	39	42.5	2500
15	13	43.3	21.4	39.2	42.7	2500
16	14	43	21.5	39.3	42.8	2500
17	15	43.4	21.5	39.5	43	2497
18	16	43.7	21.5	39.6	43	2500
19	17	43.7	21.5	39.7	43.2	2497
20	18	43.9	21.5	39.9	43.3	2500
21	19	44	21.5	40	43.4	2497
22	20	44.3	21.5	40.1	43.6	2500
23	21	44.8	21.4	40.2	43.8	2497
24	22	44.7	21.5	40.3	43.9	2497
25	23	44.9	21.5	40.4	44.1	2497
26	24	44.9	21.4	40.6	44.2	2500
27	25	45.3	21.5	40.7	44.3	2500
28	26	45.7	21.5	40.9	44.5	2500
29	27	45.8	21.4	41.1	44.7	2495
30	28	46.2	21.4	41.2	44.7	2497
31	29	46.5	21.4	41.3	44.9	2490
32	30	46.4	21.5	41.5	45	2495

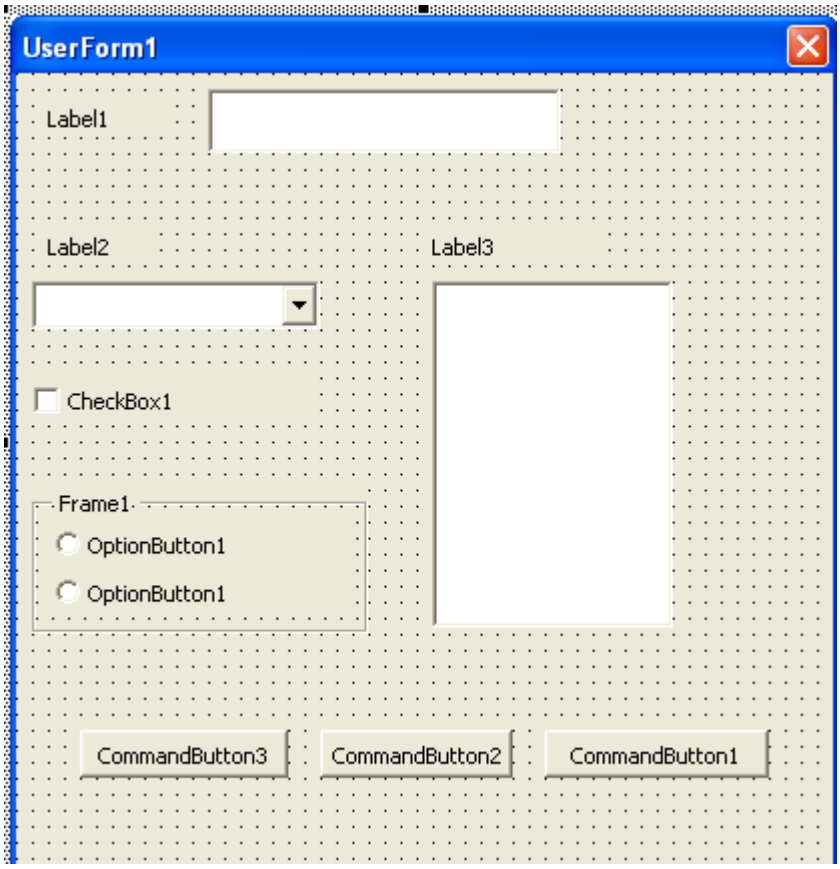
Allez dans l'éditeur VBA (Alt+F11). Dans le menu Insertion, cliquez sur UserForm. Vous allez obtenir :



Comme vous le voyez, l'objet UserForm est une boîte de dialogue vide. L'environnement de développement vous propose aussi une boîte à outil qui contient les contrôles qu'il est possible d'ajouter à la boîte de dialogue. Vous devez avoir aussi comme dans l'exemple l'affichage de la fenêtre des propriétés de l'objet UserForm. Si tel n'est pas le cas, faites la afficher en appuyant sur F4.

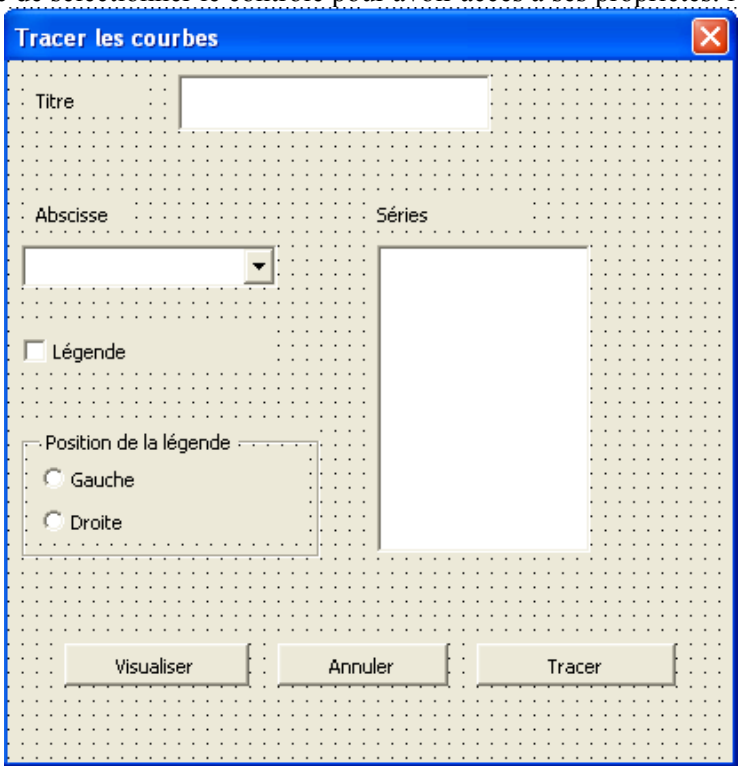
Agrandissez un peu le formulaire. Vous pouvez le faire en tirant le coin inférieur droit avec la souris, ou en modifiant les propriétés Heigth et Width.

Sur le formulaire, ajoutons un bouton. Pour faire cela, vous devez sélectionner le contrôle bouton de commande dans la boîte à outils et le tracer sur le formulaire. Ajoutons aussi, plusieurs autres contrôles pour obtenir un formulaire qui ressemble à :



Le contrôle situé sous l'intitulé 'Label3' est une zone de liste.

Nous allons commencer par modifier les textes. Pour cela il faut modifier la propriété 'Caption' des contrôles dans la fenêtre des propriétés. Celle-ci affiche les propriétés du contrôle actif, il suffit donc de sélectionner le contrôle pour avoir accès à ses propriétés. Nous voulons obtenir :



Et voila réalisé notre formulaire. Pour l'instant il ne fait rien. Pour qu'il puisse interagir avec l'utilisateur, nous allons devoir gérer son code de fonctionnement, son affichage et éventuellement sa communication.

Affichage du formulaire

Pour pouvoir afficher le formulaire, nous allons donc appeler celui-ci depuis une procédure. On déclenche l'affichage d'un formulaire en appelant la méthode Show de l'objet UserForm. Ajoutons à notre projet un module standard, et écrivons le code suivant.

```
Public Sub TraceCourbe()
    UserForm1.Show
End Sub
```

Cette méthode Show attend un argument facultatif nommé Modal qui est un booléen gérant la modalité.

Un formulaire Modal est un formulaire qui bloque l'exécution du code appelant tant que le formulaire est visible. Je n'irais pas plus loin dans le détail puisque nous ne verrons pas ici la gestion des formulaires non modaux. Si l'argument est omis, le formulaire est toujours modal.

Si vous exécutez ce code, vous voyez que votre formulaire s'affiche, mais quoique vous fassiez il ne se passe rien. Pour qu'il y ait la possibilité d'interagir, il faut écrire un code d'interaction.

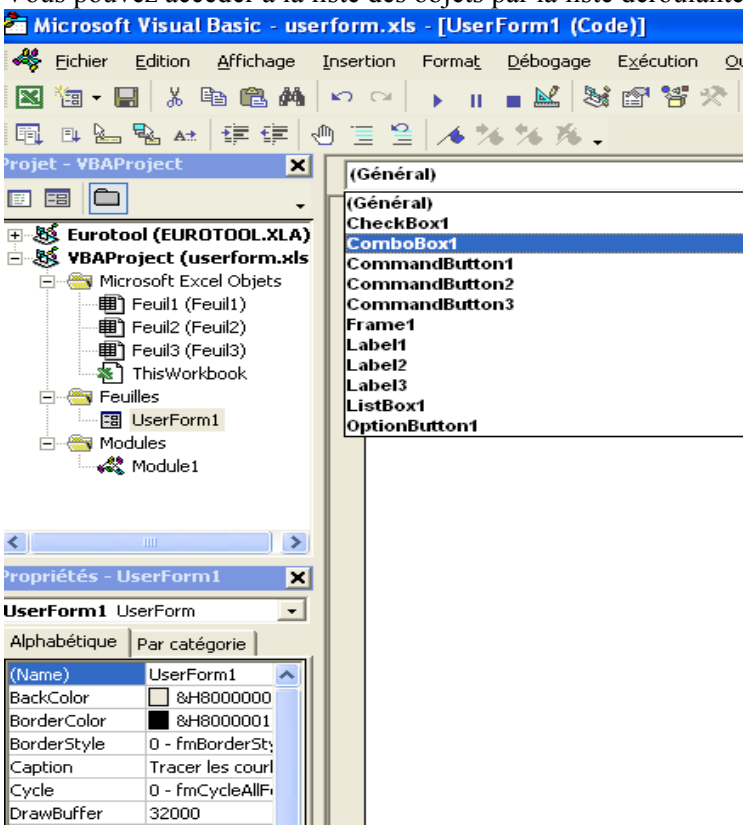
Gestion des évènements

Les contrôles sont des éléments appartenant au formulaire. Les évènements qu'ils reçoivent doivent donc être géré dans le module de code de celui-ci. Le formulaire aussi possède ses propres évènements. Commençons par gérer l'initialisation du formulaire, c'est-à-dire l'évènement qui sera levé quand le formulaire sera activé. Pour cela, retourner dans votre formulaire et appuyez sur F7 pour afficher le module de code du formulaire. Vous devriez voir apparaître :

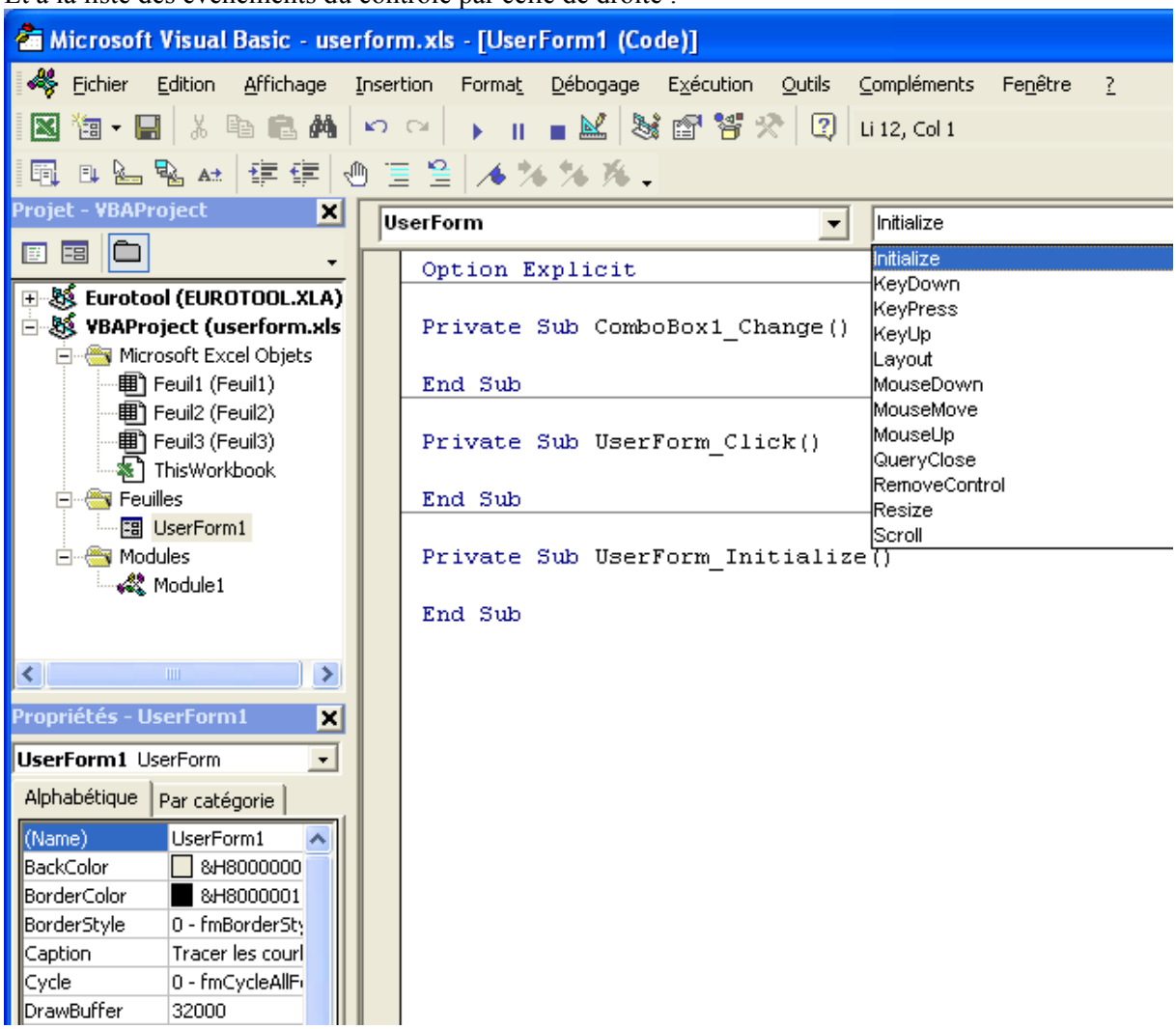
```
Option Explicit

Private Sub UserForm_Click()

End Sub
```

Par défaut l'environnement vous donne la définition de l'évènement par défaut de l'élément actif, dans notre cas l'évènement Click de l'objet UserForm.

Vous pouvez accéder à la liste des objets par la liste déroulante de gauche



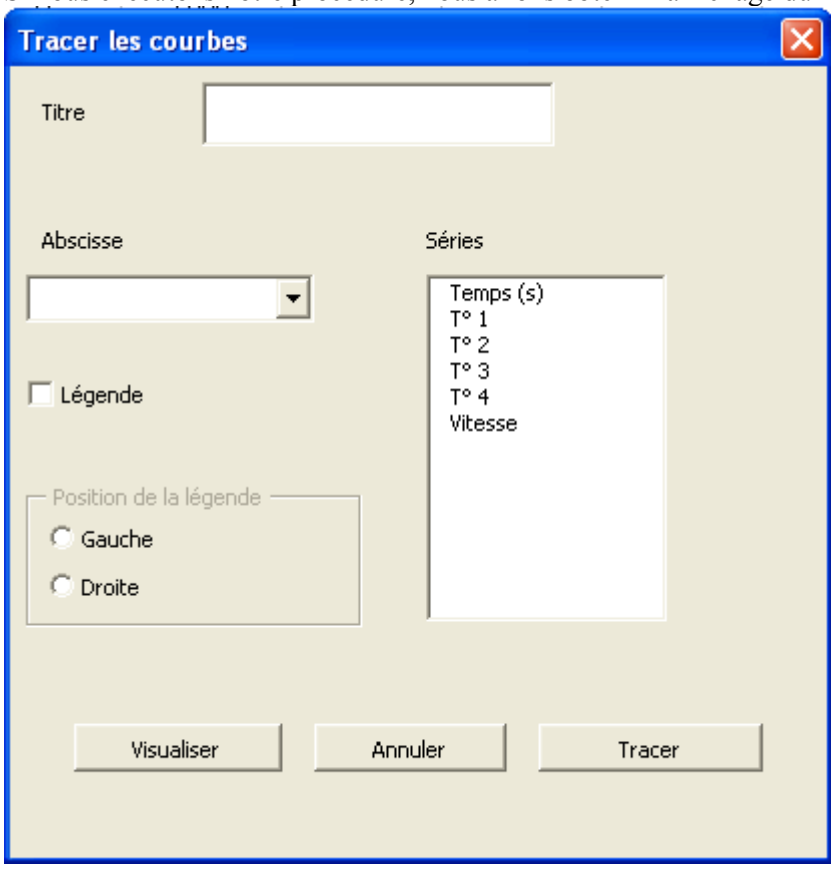
Et à la liste des évènements du contrôle par celle de droite :



Gérons donc notre évènement Initialize. Dans cet évènement nous allons écrire le code d'initialisation.

```
Option Explicit

Private Plage As Range

Private Sub UserForm_Initialize()
    Dim Feuille As Worksheet, cmpt As Long

    Set Feuille = ThisWorkbook.Worksheets(1)
    Set Plage = Feuille.UsedRange
    Me.ComboBox1.Style = fmStyleDropDownList
    Me.ListBox1.MultiSelect = fmMultiSelectMulti
    'rempli les zones de liste avec les noms des séries
    For cmpt = 1 To Plage.Columns.Count
        Me.ListBox1.AddItem Plage.Cells(cmpt).Value
        Me.ComboBox1.AddItem Plage.Cells(cmpt).Value
    Next cmpt
    'Bloque la zone d'option tant que CheckBox1 n'est pas cochée
    Me.Frame1.Enabled = False

End Sub
```

Si nous exécutons notre procédure, nous allons obtenir l'affichage du formulaire comme :

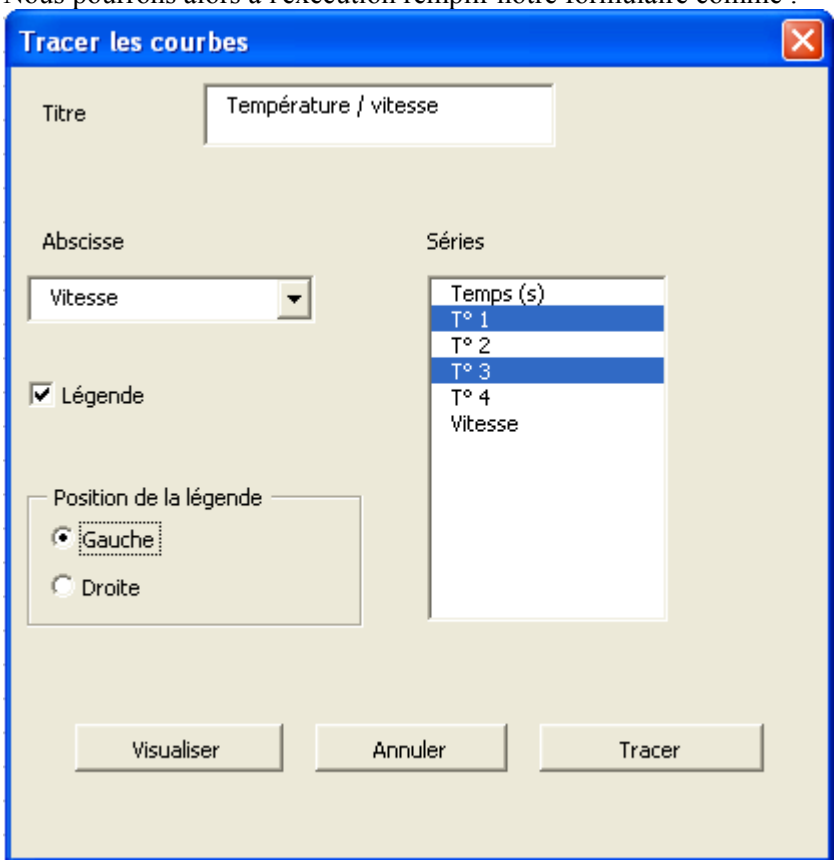


Mais pour l'instant il n'est toujours pas possible d'interagir avec le formulaire. Nous devons aussi écrire les évènements suivants :

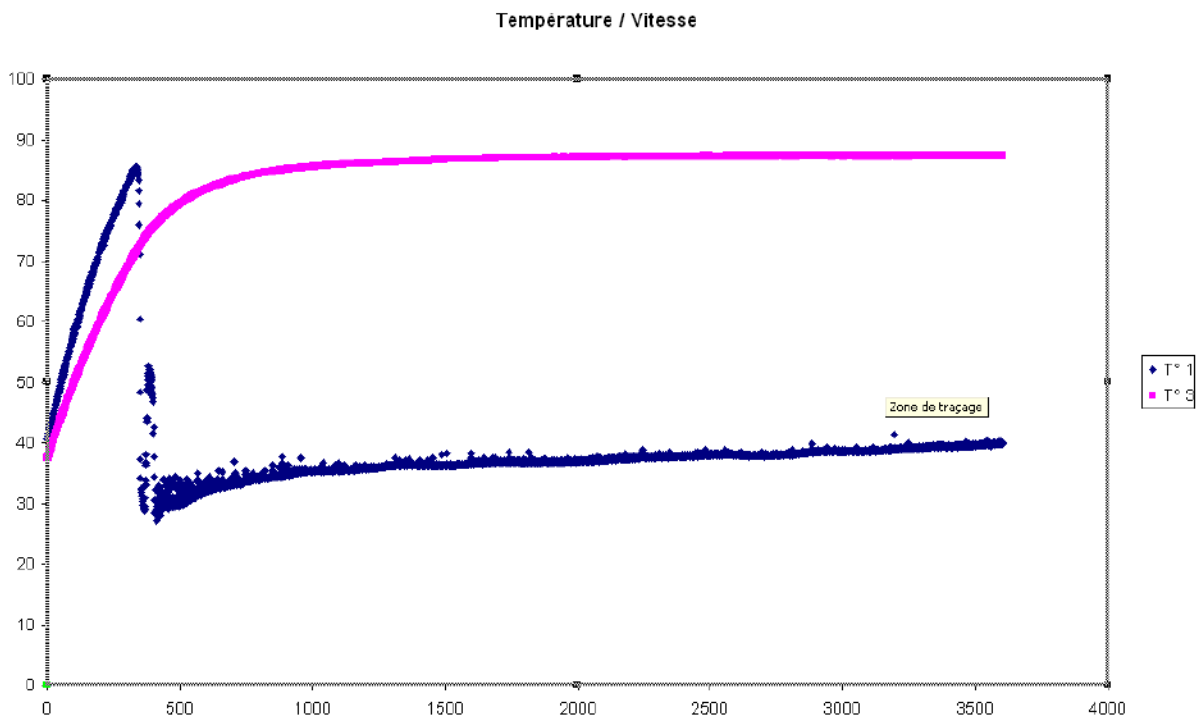
```
'affiche la zone d'option quand la case légende est cochée
Private Sub CheckBox1_Change()
        Me.Frame1.Enabled = Me.CheckBox1.Value
End Sub

'gère le bouton annuler
Private Sub CommandButton2_Click()
    Unload Me
End Sub

'gère le bouton de tracé
Private Sub CommandButton1_Click()

Dim Graphe As Chart, compteur As Long
Dim PlageX As Range, PlageY As Range, MaSerie As Series

    For compteur = 0 To Me.ListBox1.ListCount - 1
        If Me.ListBox1.Selected(compteur) Then
            If Graphe Is Nothing Then
                Set Graphe = ThisWorkbook.Charts.Add
                Graphe.ChartArea.Clear
                Graphe.ChartType = xlXYScatter
            End If
            Set PlageX = Plage.Columns(Me.ComboBox1.ListIndex + 1)
            Set PlageY = Plage.Columns(compteur + 1)
            Set MaSerie = Graphe.SeriesCollection.NewSeries
            With MaSerie
                .Values = PlageY
                .XValues = PlageX
```

```
            End With
        End If
    Next compteur
    If Not Graphe Is Nothing Then
        If Len(Me.TextBox1.Text) > 0 Then
            Graphe.HasTitle = True
            Graphe.ChartTitle.Characters.Text = Me.TextBox1.Text
        End If
        Graphe.HasLegend = Me.CheckBox1.Value
        If Graphe.HasLegend And Me.OptionButton2 Then
            Graphe.Legend.Position = xlLegendPositionLeft
        End If
    End If
    Me.Hide

End Sub
```

Nous pourrons alors à l'exécution remplir notre formulaire comme :



Et en cliquant sur 'Tracer' nous obtiendrons :



Jusque là, pas de grosse difficulté. Cependant si ce formulaire fonctionne, il est autonome, c'est-à-dire qu'il ne communique pas avec l'application appelante. En l'occurrence ce n'est pas gênant, mais supposons qu'il y ait besoin de savoir si l'utilisateur à tracer une courbe ou s'il a cliqué sur le bouton 'Annuler'.

Vous l'avez peut être oublié, mais le module de code d'un UserForm est un module objet. Cela veut dire, entre autre, qu'il est possible d'ajouter des membres à son interface. Nous allons donc lui ajouter une propriété 'Annuler' en lecture seule qui renverra Vrai si le bouton Annuler a été cliqué.

Pour écrire une propriété, on utilise l'instruction Property en lieu et place de l'instruction Sub. On doit ajouter un mot clé donnant le sans de l'action, en l'occurrence Get pour lire la propriété et Let pour l'écrire (ou Set s'il s'agit d'un objet).

Nous allons donc modifier notre code de formulaire tel que :

```
Private Plage As Range, m_Cancel As Boolean

Public Property Get Annuler() As Boolean
    Annuler = m_Cancel
End Property

'gère le bouton annuler
Private Sub CommandButton2_Click()
    m_Cancel = True
    Me.Hide
End Sub
```

Et notre code appelant ainsi :

```
Public Sub TraceCourbe()

    UserForm1.Show
    If UserForm1.Annuler = True Then
        MsgBox "tracé annuler par l'utilisateur"
    End If
    Unload UserForm1

End Sub
```

Nous pourrions utiliser un code similaire pour que le code appelant envoie des informations d'initialisation au formulaire.

Conclusion

Nous avons vu ensemble la base de la programmation VBA. Comme nous l'avons vu elle est assez simple puisque le langage ne contient pas énormément d'instruction. Tout repose en fait sur votre connaissance du modèle objet d'Excel et donc sur votre connaissance d'Excel.

Il existe bien sur de nombreuses possibilités de plus de programmation avec VBA que nous n'avons pas traité ici. Vous en trouverez de nombreux exemples sur l'excellent site de Michel Tanguy.