IFT-20403 Basic et Visual Basic

Introduction à Visual Basic pour Microsoft Excel

Mathieu Boudreault

École d'actuariat
Département d'informatique et de génie logiciel
Université Laval

20 novembre 2002

Table des matières

Programmer en Visual Basic pour Excel

Tout d'abord, il convient de faire la distinction entre Visual Basic et Visual Basic pour applications. Premièrement, il s'agit du même langage de programmation. Les commandes, les structures de programmes et l'interaction entre les objets est identique. Visual Basic est un langage complet en soi : il permet de créer des applications complètement autonomes. Visual Basic pour applications est un complément à chacun des logiciels inclus dans la suite Microsoft Office. Par exemple, Visual Basic pour Microsoft Excel permet à l'utilisateur de créer des petites applications qui interagissent avec un classeur Excel. Ces applications sont appelées macros. La principale différence entre Visual Basic et Visual Basic pour Excel réside dans les collections d'objets : les objets gérés par Visual Basic sont différents de ceux de Visual Basic pour Excel.

Objets propres à Microsoft Excel

En Visual Basic, les données nécessaires pour effectuer certains calculs sont obtenues à l'aide d'un formulaire (*Form*). Dans celui-ci, sont inclus différents objets interagissant entre eux et permettant à l'utilisateur de saisir les informations : champs de texte, boutons radio, cases à cocher, etc. (*TextBox*, *OptionBox*, *CheckBox*, etc.). En Excel, le principe est le même : au lieu de saisir les informations à l'intérieur de *TextBox*, il s'agit d'utiliser les cellules incluses dans une feuille de calcul.

Exemple simple : Je veux obtenir 2 nombres et afficher le résultat de la somme.
En VB, le code pourrait ressembler à celui-ci.

```
Formulaire.Reponse.Text = Val(Formulaire.Nombre1.Text) +
Val(Formulaire.Nombre2.Text)
```

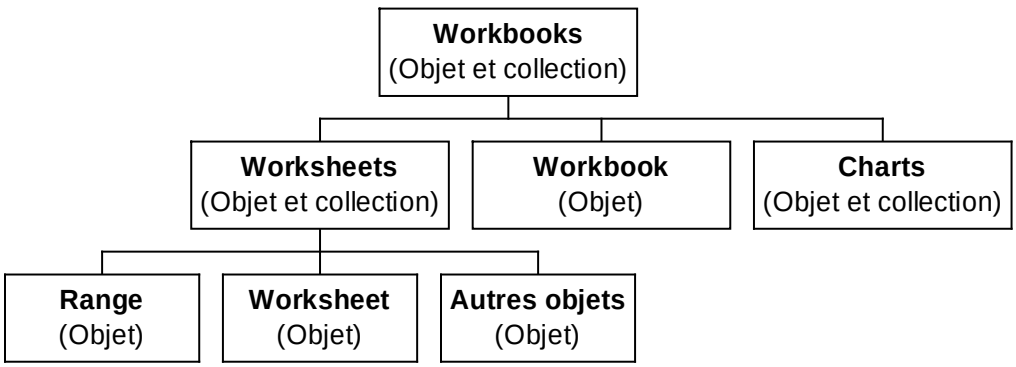
En VBA, le code ressemblerait à

```
Range("A1").Value = Range("A2").Value + Range("A3").Value
```

Note : Puisque la plupart du temps en Excel nous manipulons des chiffres, il n'est pas nécessaire d'utiliser une commande comme `Val` ou `CDbl` pour convertir du texte en nombre.

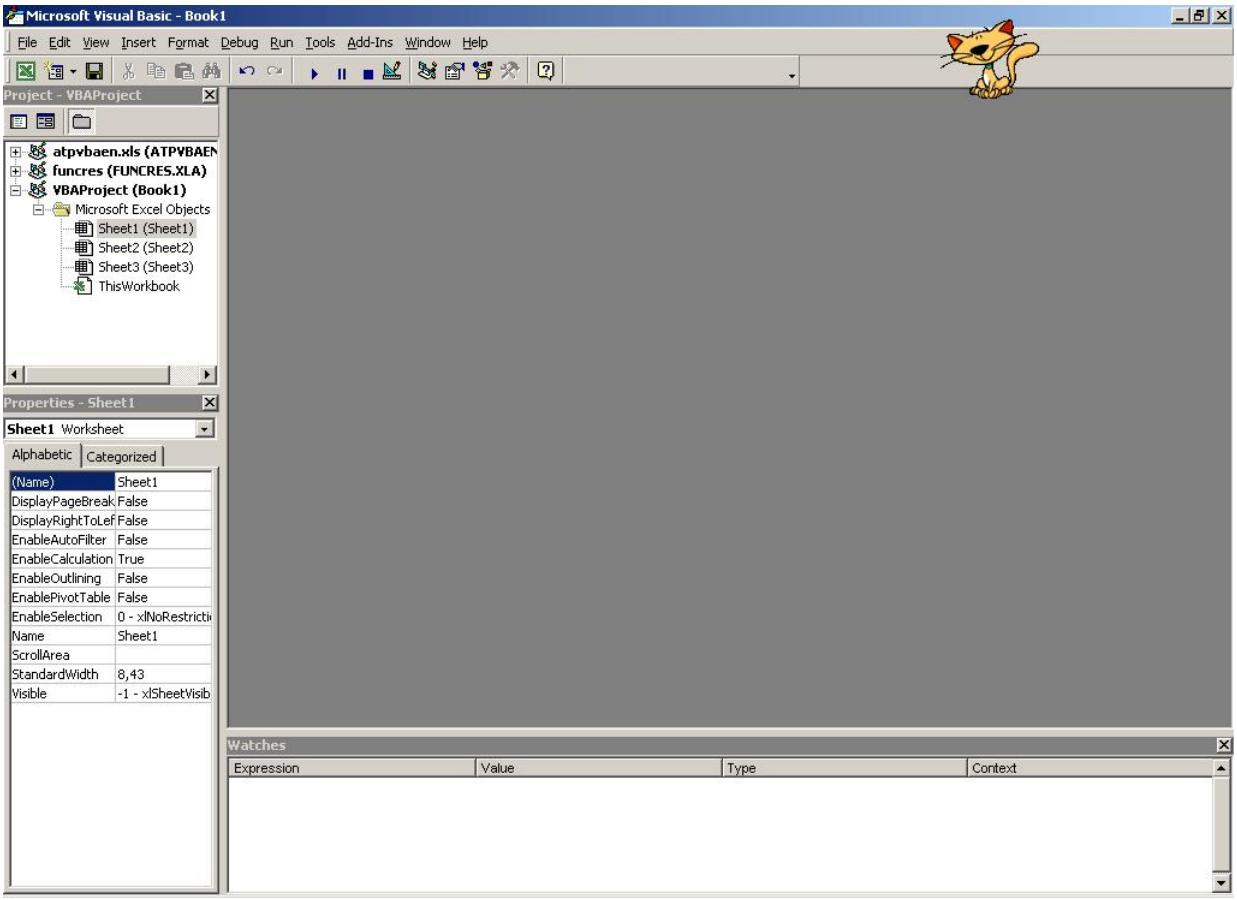
Voici une vue d'ensemble des objets et collections les plus couramment utilisés en Visual Basic pour Excel. N'oubliez pas qu'une collection est un ensemble d'objets ou d'autres collections pouvant avoir certaines propriétés et pouvant aussi interagir avec d'autres éléments.

Objets et collections
Visual Basic pour Excel



Bâtir une macro simple

À l'intérieur de Microsoft Excel, un petit sous-programme qui interagit avec les objets d'Excel est appelé une macro. Pour créer une macro par soi-même, il suffit d'aller dans le menu Outil, Macro, Éditeur Visual Basic.



Pour débuter la programmation, il suffit de sélectionner la feuille Excel dans laquelle la routine sera composée et d'aller dans le menu Affichage, Code.

Nous voulons reproduire le même tableau que dans la section précédente. Pour ce faire, nous pouvons créer une macro Excel qui fera ce boulot.

	A	B	C	D
1		**Prestation de décès**	10 000,00 $	
2				
3	**x**	**Probabilité de décès dans la prochaine année d'un individu d'âge x**	**Prime assurance temporaire un an émis à l'individu d'âge x**	
4	35	0,015	150,00 $	
5	36	0,020	200,00 $	
6	37	0,025	250,00 $	
7	38	0,030	300,00 $	
8	39	0,035	350,00 $	
9	40	0,040	400,00 $	
10				

Débutons la programmation avec la création de la routine CreerTableau.

```
Sub CreerTableau()

End Sub
```

Nous allons construire ce qui est inclus dans les cellules B1:C1. La construction du contenu des autres cellules sera expliquée un peu plus loin.

```
Sub CreerTableau()
    'Ces deux lignes servent à saisir l'information dans les cellules
    Range("B1").Value = "Prestation de décès"
    Range("C1").Value = 10000

    'Ces lignes servent à changer l'apparence du contenu des cellules
    Range("B1:C1").Font.Name = "Arial"
    Range("B1:C1").Font.Size = 12
    Range("B1:C1").Interior.ColorIndex = 34
    Range("B1:C1").Borders(xlEdgeTop).Weight = xlMedium
    Range("B1:C1").Borders(xlEdgeBottom).Weight = xlMedium
    Range("B1:C1").Borders(xlEdgeLeft).Weight = xlMedium
    Range("B1:C1").Borders(xlEdgeRight).Weight = xlMedium
    Range("B1").Font.Bold = True
    Range("C1").Style = "Currency"

    'Ces lignes servent à modifier la largeur des colonnes
    Columns("B").ColumnWidth = 25
    Columns("C").ColumnWidth = 15
End Sub
```

Il est important de comprendre le fonctionnement de l'objet *Range*. Premièrement, cet objet correspond à une plage de cellules ou à une cellule. Cet objet a certaines propriétés (*Value*, *Style*) et contient aussi certains objets (*Font*, *Interior*, *Borders*). Ces derniers ont leurs propres propriétés ou attributs (*Name*, *Size*, *Interior*, *Weight*). Remarquez aussi la hiérarchie entre les différents objets. Par exemple, la propriété *Name* appartient à l'objet *Font* qui lui appartient à l'objet *Range*. Cette hiérarchie est définie avec les points. En lisant de gauche à droite nous pouvons dire «L'objet *Range* contient l'objet *Font* qui lui a la propriété *Name* qui a été fixée à Arial».

Note : L'objet *Font* n'appartient pas qu'à l'objet *Range* ; il peut aussi servir par exemple, à l'objet *ChartTitle* qui est aussi un objet.

Gestion des cellules et des plages de cellules : objets et méthodes de base

Range

Syntaxe

```
Range("cellule")
Range("plage")
Range("cellule1,cellule2,…,celluleN")
Range("plage1,plage2,…,plageN")
```

Retourne un objet de type *Range* correspondant à la plage de cellule ou cellule correspondante.

Exemple

```
Range("A1")   ‘Stocke en mémoire le contenu de la cellule A1 et ses propriétés
Range("A1:B6")   ‘Stocke en mémoire le contenu de la plage A1:B6 et ses propriétés
Range("A1,C4,D6")   ‘Stocke en mémoire le contenu des cellules A1,C4,D6, etc.
Range("A1:B6, C3:D4")   ‘Stocke en mémoire le contenu des plages A1:B6 et C4:D4,
etc.
```

Columns ou *Rows*

Syntaxe

```
Columns("lettre de colonne")
Columns(# de colonne)
Rows(# de ligne)
```

Retourne un objet de type *Range* correspondant à la colonne spécifiée ou à la ligne spécifiée.

Exemple

```
Columns("C")   ‘Stocke en mémoire le contenu de la colonne C et ses propriétés
Columns(3)
Row(10)
Columns("C").Font.Bold = True  ‘Appliquera le style gras à toutes les cellules de
la colonne C.
```

Selection

Syntaxe

```
Selection
```

Retourne un objet de type *Range* correspondant à la cellule ou à la plage de cellules sélectionnée.

Exemple

```
Selection.Font.Bold = True   ‘Applique la mise en forme gras à la plage sélectionnée
```

ActiveCell

Syntaxe

```
ActiveCell
```

Retourne un objet de type *Range* correspondant à la cellule active. Par exemple, si la cellule A1 est active, la saisie d’information au clavier se fera alors pour la cellule A1.

Exemple

```
ActiveCell.Font.Bold = True   ‘Applique la mise en forme gras à la cellule sélectionnée
```

Offset

Syntaxe

```
ObjetRange.Offset(position relative en ligne, position relative en colonne)
```

Retourne un objet de type *Range* correspondant à la cellule positionnée de façon relative à *ObjetRange*.

Exemple

`Range("C8").Offset(-1,-2).Value` équivaut à `Range("A7").Value`

Cells

Syntaxe

```
ObjetRangeOUWorksheet.Cells(position en ligne, position en colonne)
ObjetRangeOUWorksheet.Cells(ième cellule)
```

Retourne un objet de type *Range* correspondant à la position fournie en argument.

Exemple

`Range("B1:D4").Cells(2,2).Value` équivaut à `Range("C2").Value`
`Worksheets("Feuille").Cells(1,3).Value` équivaut à `Range("C1").Value`
`Range("B1:D4").Cells(4).Value` équivaut à `Range("B2").Value`

Activate

Syntaxe

```
ObjetRangeCellule.Activate
```

Méthode qui permet d'activer une cellule. Par exemple, si la cellule A1 est active, la saisie d'information au clavier se fera alors pour la cellule A1.

Exemple

```
Range("A1").Activate
```

Select

Syntaxe

```
ObjetRange.Select
```

Méthode qui permet de sélectionner une cellule ou une plage de cellule.

Exemple

```
Range("A1:A15,C2:C26").Select
```

Remarque

Pour obtenir le contenu (valeurs ou textes) compris dans un certain objet *Range* et le stocker à l'intérieur d'une variable, il faut avoir un objet *Range* comprenant QU'UNE seule cellule.

Exemple (les prochaines lignes sont correctes)

```
Dim variable as Variant
variable = ActiveCell.Value
variable = Range("A1").Value
variable = Selection.Value ‘ Fonctionne si seulement UNE cellule est sélectionnée
variable = Range("C8").Offset(-1,-2).Value
```

Exemple (les deux dernières sont *incorrectes*)

```
Dim scalaire as Variant
Dim tableau(1 to 4,1 to 4) as Variant
scalaire = Range("A1:D4").Value
tableau = Range("A1:D4").Value
```

- La propriété *Value* ne s’applique que si l’objet *Range* correspond à UNE cellule.
- Il est quand même possible d’obtenir les valeurs contenues dans une plage de cellules avec la propriété *Cells*. Il faut par contre itérer sur toutes les lignes et les colonnes de la plage.
- Dans le cas où l’objet *Range* correspond à UNE cellule, il n’est pas nécessaire d’utiliser la propriété *Value* pour obtenir le contenu de la cellule.

Truc pour la manipulation répétitive du même objet

Lorsqu’on doit travailler souvent avec le même objet, il peut devenir très répétitif de toujours écrire la source de celui-ci. Par exemple, nous avons dû répéter à plusieurs reprises `Range("B1:C1")`. On peut par contre utiliser la structure `With`… `End With` qui permet de fixer l’utilisation d’un objet. Par exemple, la routine que nous avons conçue plus tôt aurait pu être écrite de cette façon.

```
Sub CreerTableau()

    'Ces deux lignes servent à saisir l'information dans les cellules
    Range("B1").Value = "Prestation de décès"
    Range("C1").Value = 10000

    'Ces lignes servent à changer l'apparence du contenu des cellules
    With Range("B1:C1")
        .Font.Name = "Arial"
        .Font.Size = 12
        .Interior.ColorIndex = 34
        .Borders(xlEdgeTop).Weight = xlMedium
        .Borders(xlEdgeBottom).Weight = xlMedium
        .Borders(xlEdgeLeft).Weight = xlMedium
        .Borders(xlEdgeRight).Weight = xlMedium
    End With
    Range("B1").Font.Bold = True
    Range("C1").Style = "Currency"

    'Ces lignes servent à modifier la largeur des colonnes
    Columns("B").ColumnWidth = 25
    Columns("C").ColumnWidth = 15

End Sub
```

Nous poursuivrons l'exemple du tableau après avoir regardé la programmation conditionnelle et les itérations.

Gestion des feuilles de calculs et des classeurs : objets et méthodes de base

Afin de mener à bon port la construction d'une macro quelconque, il est souvent essentiel de bien savoir manipuler les classeurs ainsi que les diverses feuilles les composant. À la fin de cette section, vous serez capable d'ouvrir des classeurs, de les enregistrer et de les fermer à l'aide des commandes Visual Basic pour Excel.

Tout d'abord, regardons les diverses commandes à exécuter pour manipuler des objets de type classeur (*Workbook*) et feuilles de calcul (*Worksheet*).

Workbooks

Syntaxe

```
Workbooks("nom du classeur")
Workbooks(# du classeur)
```

Retourne un objet de type *Workbook* (classeur) correspondant au nom de classeur spécifié. Avec un objet *Workbook* en mémoire, il est ainsi possible d'exécuter certaines méthodes applicables aux classeurs tels que son ouverture, sauvegarde, fermeture, etc.

Remarque

La spécification du nom du classeur n'est pas obligatoire (par exemple si vous voulez utiliser certaines méthodes relatives aux classeurs en général, vous verrez plus loin). Par contre, si vous devez manipuler un classeur spécifique, l'argument devient obligatoire.

Exemple

```
Workbooks("Classeur1")

‘Retourne un objet de type Workbook correspondant au classeur Classeur1. Vous
pourrez donc manipuler le classeur Classeur1 (fermer, sauvegarder, etc.)

Workbooks(1)
‘Même chose, mais correspond au premier classeur ouvert.
```

ActiveWorkbook

Syntaxe

```
ActiveWorkbook
```

Retourne un objet de type *Workbook* correspondant au classeur présentement actif (sélectionné).

Exemple

```
ActiveWorkbook.Close   ‘Ferme le classeur actif
```

Worksheets ou *Sheets*

Syntaxe

```
Worksheets("nom de la feuille de calcul")
Worksheets(# de la feuille de calcul)
Sheets("nom de la feuille de calcul")
Sheets(# de la feuille de calcul)
```

Retourne un objet de type *Worksheet* (feuille de calcul) correspondant au nom de la feuille de calcul spécifié. Avec un objet *Worksheet* en mémoire, il est ainsi possible d’exécuter certaines méthodes applicables aux feuilles de calculs.

Remarque

La spécification du nom de la feuille de calcul n’est pas obligatoire (par exemple si vous voulez utiliser certaines méthodes relatives aux feuilles en général). Par contre, si vous devez manipuler une feuille de calcul spécifique, l’argument devient obligatoire.

Exemple

```
Sheets("Feuil1")
‘Retourne un objet de type Worksheet correspondant à la feuille de calcul Feuil1

Worksheets("Donnees").Range("A1").Value
‘Permet d’obtenir le contenu de la cellule A1. La cellule A1 est un objet de type
Range. Un objet de type Range appartient à un objet de type Worksheet
(Worksheets("Donnees") permet d’obtenir un objet de type Worksheet correspondant à
la feuille Donnees). Puis, Value est une propriété associée à un objet de type
Range (Range("A1") permet d’obtenir un objet Range correspondant à la cellule A1).
```

ActiveSheet

Syntaxe

```
ActiveSheet
```

Retourne un objet de type *Worksheet* correspondant à la feuille de calcul présentement active (sélectionnée).

Exemple

```
ActiveSheet.Cells(1,1).Value
‘Retourne le contenu de la cellule A1 de la feuille active.
```

Ouverture d’un classeur *Excel* – Méthode *Open*

Il est parfois très utile de pouvoir ouvrir un classeur *Excel* afin de pouvoir lire l’information contenue dans ce classeur.

Syntaxe

```
ObjetWorkbook.Open Filename:="chemin complet du fichier Excel"
```

Exemple

Voici la commande qui vous permettra d’ouvrir un classeur *Excel* :

```
Workbooks.Open Filename:="C:\Mes documents\Classeur.xls"
```

La commande ci-haut vous permettra d’ouvrir le classeur nommé *Classeur* dans le dossier *Mes documents* sur le disque *C*.

Création d’un nouveau classeur *Excel* – Méthode *Add*

Voici la commande qui vous permettra d’ouvrir un nouveau classeur *Excel* ne contenant aucune information :

Syntaxe

```
Workbooks.Add
```

Sélection (ou activation) d’un classeur ou d’une feuille de calcul – Méthodes *Activate* et *Select*

Afin de pouvoir accéder à l’information contenue dans une feuille ou dans un classeur donné, il est essentiel de sélectionner la feuille ou le classeur dans lequel les commandes exécutées s’appliqueront.

Syntaxe - Activation d'un classeur

```
ObjetWorkbook.Activate
```

Exemple

```
Workbooks("Classeur1").Activate
```

Syntaxe - Activation et sélection d'une feuille de travail

```
ObjetWorksheet.Activate
ObjetWorksheet.Select
```

Exemple

```
Sheets("feuil1").Select
```

Sauvegarde d'un classeur *Excel* – Méthodes *SaveAs* et *Save*

Après avoir modifier l'information contenue dans un classeur, il peut être utile de vouloir sauvegarder cette information.

Syntaxe - Sauvegarder un nouveau fichier

```
ObjetWorkbook.SaveAs Filename:=”nom complet du nouveau fichier Excel”
```

Exemple

```
ActiveWorkbook.SaveAs Filename:="C:\Mes documents\classeur1.xls"
```

Cette méthode vous permet de sauvegarder le classeur actif sous le nom *classeur1.xls* dans le dossier *Mes documents* sur le disque *C*.

Syntaxe - Sauvegarder un fichier existant

```
ObjetWorkbook.Save
```

Exemple

```
Workbooks("Classeur1").Save
```

Fermeture d'un classeur *Excel* – Méthode *Close*

Fermer les classeurs non utilisés est une bonne habitude de programmation puisqu'elle permet de réduire le nombre de fichiers actifs.

Syntaxe

```
ObjetWorkbook.Close
```

Exemple

Voici la commande qui vous permettra de fermer un classeur dont l'utilisation n'est plus requise.

```
Workbooks("Classeur1").Close
```

Déclaration de variables

Rappel provenant des sections précédentes

Variable : Espace en mémoire réservé pour y stocker de l'information. Une variable peut être de plusieurs types : numérique (entier, réel), booléen (ou logique), texte (ou chaîne de caractères, aussi appelé *string*).

Dans l'utilisation des variables, il y a toujours deux étapes nécessaires. Premièrement, la déclaration qui alloue ou réserve un espace en mémoire. Deuxièmement, le stockage utilise l'espace qui lui est réservé pour conserver l'information voulue.

Déclaration de variables scalaires

Une variable scalaire est une variable comprenant qu'UNE seule information. Cette information peut être de plusieurs types.

Syntaxe

Variables dont le contenu est conservé à l'intérieur d'une procédure ou d'une fonction (portée de niveau procédure)

```
Dim scalaire as Type
Dim scalaire1, scalaire2, …, scalaireN as Type
Dim scalaire1 as Type1, scalaire2 as Type2, …, scalaireN as TypeN
```

Variables dont le contenu est conservé tant et aussi longtemps que le programme principal n'est pas interrompu (portée de niveau module)

```
Private scalaire as Type
Private scalaire1, scalaire2, …, scalaireN as Type
Private scalaire1 as Type1, scalaire2 as Type2, …, scalaireN as TypeN
```

Remarque

Les variables à portée de niveau module doivent être déclarées avant toutes les procédures du module.

Types de données

(Tableau tiré du livre Excel 2000 et Visual Basic pour Applications 6)

Types de données	*Valeurs acceptées*	*Mémoire occupée*
`Byte`	Nombre entier compris entre 0 et 255	1 octet
`Integer`	Nombre entier compris entre -32768 et 32768	2 octets
`Long`	Nombre entier compris entre –2147483648 et 2147483647	4 octets
`Simple`	Nombre à virgule flottante compris dans ±3.403E38	4 octets
`Double`	Nombre à virgule flottante compris dans ±1.7977E308	8 octets
`Currency`	Nombre à virgule fixe avec quinze chiffres pour la partie entière et quatre chiffres pour la partie décimales compris dans ±922337203685477.5808	8 octets

Il existe aussi les types de données *String* (chaînes de caractères), *Date*, etc.

Exemple

```
Private salaire as Double 'Variable à portée de niveau module

Sub calculeBonus()
      Dim bonus as Double 'Variable à portée de niveau procédure
End Sub
```

Déclaration de variables matrices

Une variable matrice est une variable comprenant plusieurs séries d'informations. Elle peut être comparée à un tableau ou à une matrice. Par contre, en Visual Basic tout comme dans tout autre langage de programmation, la variable matrice peut avoir plus que deux dimensions.

Syntaxe

Variables à dimensions définies

```
Dim Tableau(bi1 to bs1,bi2 to bs2,…, biN to bsN) as Type
Private Tableau(bi1 to bs1,bi2 to bs2,…, biN to bsN) as Type

Dim Tableau(n1,n2,…, nN) as Type
Private Tableau(n1,n2,…, nN) as Type
```

Variables à dimensions indéfinies

```
Dim Tableau() as Type
Private Tableau() as Type
```

Légende

- `bi1` ➔ représente l'adresse de la borne inférieure de la 1ère dimension du tableau.
- `bs1` ➔ représente l'adresse de la borne supérieure de la 1ère dimension du tableau.
- `n1` ➔ représente le nombre d'éléments dans la 1ère dimension du tableau.

Remarques

- Les `bi1, bs1, n1, etc.` doivent être des nombres entiers entrés par le programmeur. Ils ne peuvent représenter le contenu d'une variable.
- Si aucune adresse n'est définie, Visual Basic utilisera celle spécifiée par `Option Base`. Si la ligne `Option Base` n'est pas présente et que les adresses des bornes ne sont pas définies, les bornes iront de `0 jusqu'à ni-1`
- Si les dimensions d'une variable tableau ne sont pas définies, aucun contenu ne peut y être stocké tant et aussi longtemps que Visual Basic ne connaît pas ses dimensions. Vous pouvez redéfinir les dimensions d'une variable tableau à l'aide de la commande *ReDim* (voir plus loin).

Exemple # 1

À quoi pourraient servir les adresses des variables matrice ? Disons que vous simulez la performance d'un indice boursier entre 2003 et 2050. Déclarer la variable de cette façon…

```
Dim IndicesBoursiers(2003 to 2050) as Double
```

… et obtenir la simulation de l'année 2023…

```
SimulationVoulue = IndicesBoursiers(2023)
```

... est beaucoup plus simple que…

```
Dim IndicesBoursiers(48) as Double
SimulationVoulue = IndicesBoursiers(21)
```

Exemple # 2

```
Option Base 1
```

```
Sub NomRoutine()
      Dim Tableau4dimensions(10, 20, 30, 40) as Integer
End Sub
```

est équivalent à

```
Sub NomRoutine()
      Dim Tableau4dimensions(1 to 10, 1 to 20, 1 to 30, 1 to 40) as Integer
End Sub
```

Assigner et obtenir des valeurs d'une variable matrice

L'assignation d'une valeur à un élément d'une matrice se fait à peu près de la même façon qu'avec une variable scalaire (standard). Il faut évidemment spécifier la position dans la matrice où il faut stocker la donnée.

Exemple

```
Dim TableStandard(1 to 50, 1 to 25) as Double   'Déclaration de la variable matrice
TableStandard(2,3) = 1434   'Stocker la valeur 1434 dans la 2e ligne, 3e colonne de
la matrice Table standard.

Dim Tableau4dimensions(10, 20, 30, 40) as Integer   'Déclaration de la variable
matrice
Tableau4dimensions(1,1,1,1) = 1536   'Stocker la valeur 1536 à la position
(1,1,1,1) de la variable matrice.
```

L'affichage du contenu d'une variable matrice se fait de façon similaire.

Exemple (suite)

```
Range("C1").Value = TableStandard(2,3)   'Affiche dans la cellule C1 le contenu de
la variable TableStandard à la ligne 2, colonne 3.
```

Redim

Nous avons mentionné plus tôt que les dimensions d'une variable matrice doivent être spécifiées explicitement (nombres entrés directement par le programmeur) ou pas spécifiées du tout. L'instruction *Redim* permet de redimensionner une variable matrice. Contrairement à l'instruction *Dim*, *Redim* peut accepter le contenu d'une variable.

Syntaxe

```
ReDim Tableau(bi1 to bs1,bi2 to bs2,…, biN to bsN) as Type
ReDim Tableau(n1,n2,…, nN) as Type
```

Exemple

```
Dim nbresim as Integer
Dim Simulations() as Double
nbresim = 1000
Redim Simulations(2003 to 2003 + nbresim - 1) as Double
'car Dim Simulations(2003 to 2003 + nbresim - 1) as Double est illégal
```

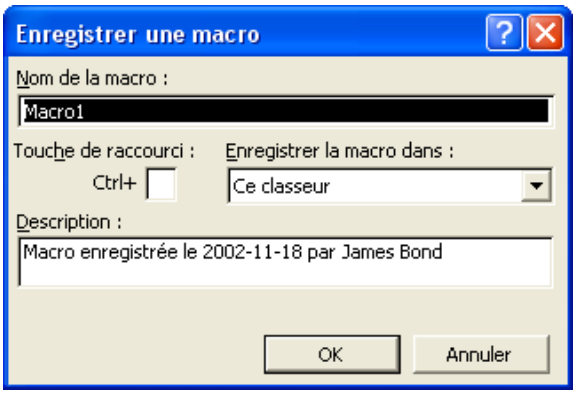
Création de routines et de fonctions personnelles

Il faut tout d'abord faire la distinction entre une routine et une fonction. Premièrement, une routine est un sous-programme qui exécute certaines commandes. Une fonction exécute aussi certaines commandes, mais en plus, retourne une certaine valeur ou tableau de valeurs.

Enregistrer une macro

Il existe plusieurs façons de créer des routines en VBA. L'une d'entre elles utilise la boîte de dialogue « Enregistrer une macro ». Cette méthode a l'avantage de ne nécessiter aucune programmation par l'utilisateur ; celui-ci n'a qu'à démarrer l'enregistrement de la macro, exécuter ce qui doit être emmagasiné dans la macro, et arrêter l'enregistrement. Cette façon de faire est utilisée lorsque la macro consiste en une procédure courante et répétitive que l'on a à faire, et que l'on désire automatiser. Voici comment procéder :

1. Dans le menu Outils de Excel, sélectionnez Macro, Nouvelle macro. La boîte de dialogue Enregistrez une macro s'affiche :



Il s'agit maintenant de donner un nom à la macro sous Nom de la macro. Il est possible d'attribuer une touche de raccourci à cette macro. Généralement, l'enregistrement de la macro se fait dans Ce classeur. Il est possible d'entrer une brève description de la macro.

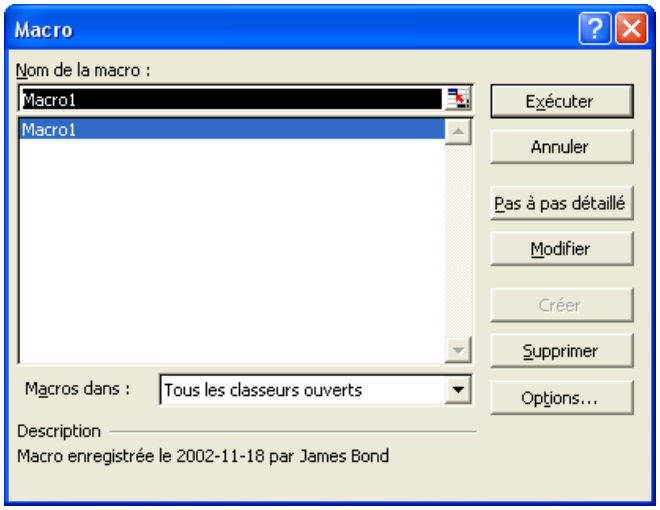
2. Appuyez sur OK. La barre d'outils Arrêt de l'enregistrement s'affiche, indiquant que l'enregistrement de la macro est en cours.

Il ne reste plus qu'à exécuter (dans l'ordre) tout ce qui doit se retrouver dans la macro. Par exemple, quelqu'un pourrait écrire le chiffre 28 dans la cellule A1, changer le format de cellule en « monétaire » pour obtenir 28,00 $ (bouton droit sur la cellule, Format de cellule, sélection de Monétaire sous l'onglet Nombre, ok), mettre la cellule en gras, pour ensuite arrêter l'enregistrement en appuyant sur le bouton carré de la barre d'outils Arrêt de l'enregistrement.

Une fois la macro enregistrée, il est possible de l'utiliser.

3. Effacez tout ce qui se trouve dans la cellule A1 (Édition, Effacer, tout).

4. Choisissez Macro, puis Macros… du menu Outils. La boîte de dialogue Macro s'affiche, vous permettant d'aller chercher la macro créée.



5. Appuyez sur Exécuter. La macro s'exécute, et refait ce que vous avez entré au préalable.

Il est possible d'aller voir la programmation qui a été nécessaire à la création de cette macro, et ce de deux façons. Vous pouvez retourner à la boîte de dialogue Macro (montrée ci-haut), et sélectionner Modifier. L'éditeur Visual Basic s'ouvrira et montrera les codes. Vous pouvez également tout simplement ouvrir Visual Basic en cliquant sur Macro, Visual Basic Editor du menu Outils.

```
Sub Macro1()
'
' Macro1 Macro
' Macro enregistrée le 2002-11-18 par James Bond
'

'
    Range("A1").Select
    ActiveCell.FormulaR1C1 = "28"
    Range("A1").Select
    Selection.NumberFormat = "#,##0.00 $"
    Selection.Font.Bold = True
End Sub
```

Finalement, il est possible d'insérer un bouton dans la feuille Excel qui, lorsqu'on appuie dessus, exécute une macro.

1. Sous le menu Affichage, sélectionnez Barre d'outils, Formulaires. Une boîte à outils apparaît, que vous pouvez insérer dans la Barre d'outils et de menus en haut de l'écran.
2. Cliquez sur Bouton, représenté par un petit bouton gris. Votre curseur devient une croix. Il vous suffit maintenant de dessiner un bouton de la taille désirée.

Excel ouvre ensuite automatiquement la boîte de dialogue Affecter une macro. Vous pouvez maintenant sélectionner la macro que vous venez de créer et appuyer sur OK. Le bouton est maintenant fonctionnel, et exécutera la macro lorsque l'utilisateur appuie dessus. À noter qu'il est possible de changer le texte apparaissant sur le bouton, en cliquant dessus avec le bouton droit, puis le gauche. **Note importante** : Les macros que vous affectez à un bouton ne doivent pas recevoir d'arguments obligatoires.

Création de routines personnelles

Il a été mentionné plus tôt qu'il était préférable de diviser un problème en plusieurs petits problèmes. Le déboguage s'en trouve ainsi simplifié. Une façon d'y parvenir est de créer une routine qui exécutera certaines commandes. Par exemple, une routine peut représenter une étape importante d'un algorithme.

Syntaxe

```
[Private ou Public] Sub NomDeLaRoutine(argument1 as Type1, argument2 as Type2,…)
      Instructions
End Sub
```

Remarques

- Un peu comme une variable, la routine peut avoir une portée privée ou publique, c'est-à-dire être disponible à un certain module Excel ou à tous les modules. Si le mot `private` ou `public` est omis, la routine sera publique.
- Il n'est pas nécessaire de spécifier des arguments à la routine. Vous pouvez aussi spécifier des arguments optionnels avec des valeurs par défaut. Voir la façon de procéder dans les fonctions personnelles, plus loin.
- Il est également possible d'assigner à un bouton une procédure créée de cette façon si la routine n'a aucun argument obligatoire.

Exemple

```
‘Routine simple, de portée publique, sans arguments.
Sub Programme()
      Instructions
End Sub

‘Routine plus complexe, de portée privée, avec argument.
Private Sub Initialisation(NomFichier as String)
      Instructions
End Sub
```

Appel de routines personnelles

Il y a deux façons d’appeler des routines personnelles dans Visual Basic pour Excel.

Syntaxe

```
‘Première façon, Instruction Call

Call NomDeLaRoutineSansArgument
Call NomDeLaRoutineAvecArguments(Arg1, Arg2, …)

‘Deuxième façon
NomDeLaRoutineSansArgument
NomDeLaRoutineAvecArguments Arg1, Arg2,…
```

Exemple

```
‘Première façon, Instruction Call
Call Programme
Call Initialisation("autoexec.bat")

‘Deuxième façon
Programme
Initialisation "autoexec.bat"
```

Création de fonctions personnelles

Une autre façon de faciliter le déboguage est l’utilisation de fonctions personnelles qui permet de vérifier certains calculs intermédiaires ou d’exécuter des calculs souvent exécutés dans le programme. Une fonction est différente d’une routine car elle retourne une valeur.

Syntaxe de la création de fonction

```
[Public ou Private] Function NomDeLaFonction(arguments) as Type
      Instructions
      NomDeLaFonction = expression
End Function
```

Syntaxe des arguments (`arguments`)

```
NomArgument As Type
[Optional] NomArgument As Type =(Valeur par défaut)
```

Remarques

- Un peu comme une variable, la fonction peut avoir une portée privée ou publique, c'est-à-dire être disponible à un certain module Excel ou à tous les modules. Si le mot `private` ou `public` est omis, la routine sera privée.
- Pour que la fonction puisse retourner une valeur, on doit absolument retrouver au moins une ligne de code ayant `NomDeLaFonction = expression`. Sinon la fonction retourne 0 et est inutile.
- La déclaration des arguments se fait la plupart du temps de la première façon, c'est-à-dire, `NomArgument As Type`.
- Un argument peut aussi être optionnel, c'est-à-dire que l'utilisateur pourrait appeler la fonction de façon correcte sans spécifier de valeur pour cet argument. Par contre, une valeur par défaut doit être présente. Une valeur par défaut est une valeur utilisée par un programme si l'utilisateur n'en spécifie aucune.
- Les arguments optionnels doivent être déclarés à la fin.
- La structure des arguments présentée dans cette section s'applique aussi aux routines.

Exemple

Voici la création d'une fonction avec un argument qui spécifie une valeur par défaut. Nous verrons comment appeler des fonctions avec des arguments par défaut et la différence. Plus bas, la même fonction est présentée, sans argument optionnel.

```
'Cet exemple, extrêmement simplifié de la réalité, calcule la cotisation à la RRQ.
'L'argument taux est optionnel
Function CotisationRRQAvecTauxOpt(salaire As Double, Optional taux As Double =
0.035) As Double
    CotisationRRQAvecTauxOpt = salaire * taux
End Function

'L'argument taux n'est PAS optionnel
Function CotisationRRQSansTauxOpt (salaire As Double, taux As Double) As Double
    CotisationRRQSansTauxOpt = salaire * taux
End Function
```

Appel de fonctions personnelles

Lorsque nous avons besoin du résultat d'un calcul complexe qui est heureusement, exécuté à l'intérieur d'une fonction personnelle, nous utiliserons la façon suivante pour appeler une fonction.

Exemple

Je voudrais calculer ma cotisation au régime des rentes du Québec avec mon salaire.

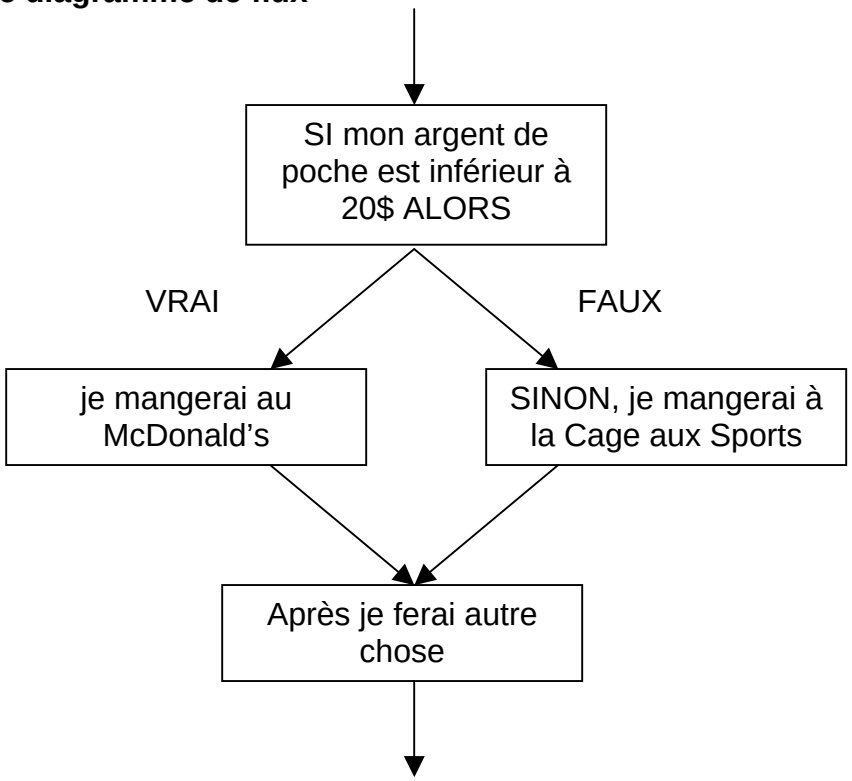
```
Dim MonSalaire, MaCotisation as Double
MonSalaire = 45000

'Appel de la fonction avec taux OPTIONNEL
MaCotisation = CotisationRRQAvecTauxOpt(MonSalaire)
'Il y a 1575 dans la variable MaCotisation.

'Appel de la fonction avec taux OPTIONNEL
```

```
MaCotisation = CotisationRRQAvecTauxOpt(MonSalaire, 0.04)
‘Il y a 1800 dans la variable MaCotisation.

‘Appel de la fonction SANS taux OPTIONNEL
MaCotisation = CotisationRRQSansTauxOpt(MonSalaire, 0.04)
‘Il y aura toujours 1800 dans la variable MaCotisation.

‘Appel de la fonction SANS taux OPTIONNEL
MaCotisation = CotisationRRQSansTauxOpt(MonSalaire)
‘Retourne une erreur car le second argument n’est pas optionnel.
```

Programmation conditionnelle

La programmation conditionnelle permet d’exécuter certaines commandes selon la réalisation de certains événements ou l’obtention de certains résultats. Pour mieux comprendre les différentes possibilités, le programmeur peut utiliser un diagramme de flux. Vous remarquerez que la programmation conditionnelle à l’intérieur de Visual Basic pour applications est la même qu’à l’intérieur de Visual Basic.

Exemple de diagramme de flux



Toute programmation conditionnelle nécessite la construction d’une condition. Que ce soit en Visual Basic ou tout autre langage, une condition est formée par une ou plusieurs comparaisons. On forme une comparaison à l’aide des opérateurs de comparaison et on relie les comparaisons avec les opérateurs logiques.

Créer une condition

Avant tout, définissons les opérateurs de Visual Basic pour applications.

Opérateurs de comparaison

<	Strictement inférieur
>	Strictement supérieur
=	Est exactement égal à
<=	Inférieur ou égal
>=	Supérieur ou égal
<>	Est différent de

Opérateurs logiques

Les opérateurs logiques servent à unir les comparaisons pour former une condition.

And	ET (intersection)
Or	OU (union)

Notons tout d'abord `OPcomp` pour opérateur de comparaison et `OPlog` pour opérateur logique.

Syntaxe

```
Element1 OPcomp Element2 OPlog Element3 OPcomp Element4 OPlog…
```

Remarque

L'utilisation de parenthèses permet de déterminer un certain ordre entre les conditions.

Exemple

```
‘Ce programme détermine si un étudiant va être engagé dans la firme d’actuaires ABC
Dim nbreexams as Integer
Dim moyenne as Double
Dim cyclesuperieur as Boolean
nbreexams = 3
moyenne = 3.23
cyclesuperieur = False

‘Exemples de condition pour engager un étudiant (n’est pas basé sur la réalité… exemple fictif)
‘L’employeur veut que l’étudiant ait au moins 2 examens professionnels ET une moyenne cumulative d’au moins 3. La condition est donc :
‘nbreexams >= 2 And moyenne >= 3   ‘Retourne vrai car VRAI ET VRAI = VRAI
‘L’employeur considère qu’un diplôme de cycle supérieur équivaut aux deux examens. La condition est donc :
‘(nbreexams >= 2 Or cyclesuperieur = True) And moyenne >= 3   ‘Retourne vrai car (VRAI OU FAUX = VRAI) ET VRAI = VRAI

nbreexams = 1
```

```
cyclesuperieur = True
moyenne = 3.51

‘L’employeur veut que l’étudiant ait au moins 2 examens professionnels ET une
moyenne cumulative d’au moins 3. La condition est donc :
‘nbreexams >= 2 And moyenne >= 3   ‘Retourne faux car FAUX ET VRAI = FAUX
‘L’employeur considère qu’un diplôme de cycle supérieur équivaut aux deux examens.
La condition est donc :
‘(nbreexams >= 2 Or cyclesuperieur = True) And moyenne >= 3   ‘Retourne vrai car
(FAUX OU VRAI = VRAI) ET VRAI = VRAI
```

Programmation conditionnelle simple

Syntaxe

```
If condition Then
      commandes si condition est vraie (1)
Else
      commandes si condition est fausse
End If
```

Remarques

- Peu importe la complexité de la condition, celle-ci doit être vraie pour exécuter les commandes en (1).
- Il est aussi possible d’imbriquer plusieurs blocs `If`.

Syntaxe

```
If condition1 Then
   commandes si condition1 est vraie
         If condition2 Then
               commandes si condition1 ET condition2 sont vraies
         Else
               commandes si condition1 est vraie ET condition2 est fausse
         End If
Else
   commandes si condition1 est fausse
         If condition3 Then
               commandes si condition1 est fausse ET condition3 est vraie
         Else
               commandes si condition1 ET condition3 sont fausses
         End If
End If
```

- On peut aussi laisser tomber le `Else` dans le cas où le programme ne devrait rien faire si la condition n’est pas respectée.

Programmation conditionnelle par cas

Syntaxe

```
Select Case variable
      Case condition1partielle
            commandes si condition1partielle est vraie
      Case condition2partielle
            commandes si condition2partielle est vraie
      …
      Case Else
            Commandes si aucune condition partielle n'a été respectée
End Select
```

Remarques

Une condition partielle peut être :

- `nombre` si la comparaison équivaut à `variable = nombre`
- `nombre1,nombre2` si la comparaison équivaut à `variable = nombre1 Or variable = nombre2`
- `nombre1 to nombreN` si la comparaison équivaut à `variable = nombre1 Or variable = nombre2 Or … Or variable = nombreN`
- `Is OPcomp nombre` si la comparaison équivaut à `variable OPcomp nombre`

Un exemple viendra éclaircir le tout.

(Exemple tiré de l'aide de Visual Basic)

```
Select Case performance
    Case 1 'si performance=1
        Bonus = Salaire * 0.1
    Case 2, 3 'si performance = 2 ou 3
        Bonus = Salaire * 0.09
    Case 4 To 6 'si performance = 4 ou 5 ou 6
        Bonus = Salaire * 0.07
    Case Is > 8 'si performance > 8
        Bonus = 100
    Case Else 'si performance = 7 ou autre valeur
        Bonus = 0
End Select
```

Programmation itérative

Il arrive très fréquemment en programmation que certaines instructions doivent être répétées constamment. Le nombre de répétitions peut soit être fixée d'avance par une variable ou constante, ou le nombre de répétitions peut être conditionnel à l'évolution d'une certaine condition. Dans le premier cas, on utilisera les itérations simples, dans le second, les itérations conditionnelles.

Itération simple (*For… Next*)

Lorsque le nombre d'itérations à exécuter est fixé à l'intérieur du programme, que ce soit par une constante ou une variable, il est préférable d'utiliser une forme d'itération simple.

Syntaxe

```
For compteur = borneinf to bornesup
      Instructions
Next compteur
```

Remarques

- L'itération illustrée dans la syntaxe produira (`bornesup-borneinf+1`) itérations.
- Pour répéter `n` fois une certaine instruction, la forme la plus commune d'itération simple est
  ```
  For compteur = 1 to n
        Instructions
  Next compteur
  ```
- Il est également possible d'utiliser un compteur avec un pas différent de 1. En effet, si on inscrit `For compteur = borneinf to bornesup Step s`, la variable `compteur` prendra comme valeurs : `borneinf, borneinf +s, borneinf +2s, etc`. et l'itération arrêtera une fois que la borne supérieure sera rencontrée.

Itération conditionnelle (*While… Wend*)

Lorsque nous voulons qu'une série d'instructions s'exécute tant et aussi longtemps qu'une certaine condition est respectée (reste vraie), nous utiliserons l'itération conditionnelle.

Syntaxe

```
Instructions d'initialisation
While condition
      Instructions si condition est vraie
Wend
```

Remarques

- Les instructions d'initialisation sont celles qui permettent d'entrer dans la boucle et celles qui donnent une certaine information (tel qu'un compteur).
- Un des éléments formant la condition DOIT se mettre à jour à l'intérieur de l'itération.
- La condition est bâtie de la même façon qu'à l'aide de la programmation conditionnelle.
- Si vous voulez utiliser un compteur, vous devrez le créer (en déclarant une variable) et l'incrémenter vous-mêmes à l'intérieur de la boucle.
- Attention aux boucles à l'infini ! Assurez-vous qu'il est possible que la condition passe de vrai à faux un jour !

Gestion des boîtes de dialogue

Lorsque nous voudrons obtenir de l'information d'un utilisateur, nous utilisons une boîte de dialogue appelée *InputBox*. La boîte *InputBox* est utilisée quand l'information transmise peut être du texte, un nombre, etc. Toutefois, lorsque nous voulons communiquer de l'information à l'utilisateur ou obtenir une réponse courte (oui ou non), nous utiliserons une boîte de type *MsgBox*.

Boîte de dialogue *InputBox*

Au lieu d'utiliser des cellules dans une feuille Excel afin d'obtenir des données de l'utilisateur, il peut être plus convenable d'utiliser une boîte *InputBox*.

Syntaxe

```
Variable = InputBox("question posée","titre de la boite","valeur par défaut")
```

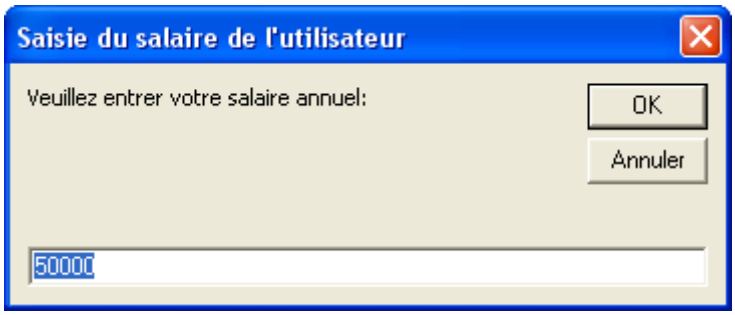
Arguments

`"question posée"`	est le texte qui apparaîtra dans la boîte de dialogue, qui est généralement une question posée à l'utilisateur.
`"titre de la boite"`	est facultatif et représente le titre de la boîte de dialogue; s'il est omis, le nom de l'application apparaîtra.
`"valeur par défaut"`	est lui aussi facultatif, correspond au texte apparaissant par défaut dans la zone de texte lors de l'affichage de la boîte de dialogue.

Exemple

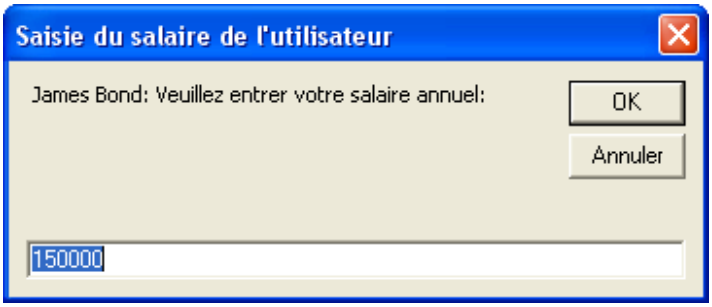
Admettons que l'on veuille obtenir le salaire annuel de l'utilisateur. Cette procédure stockerait le salaire obtenu de l'utilisateur dans la variable « salaire » :

```
Sub Salaire_annuel()

Dim salaire As Single

salaire = InputBox("Veuillez entrer votre salaire annuel:", _
"Saisie du salaire de l'utilisateur", 50000)

End Sub
```



À noter qu'il est possible d'insérer des variables dans la fonction *InputBox* avec l'opérateur de concaténation « & ». Par exemple, si le nom de l'utilisateur est inscrit dans la cellule A1, on pourrait avoir le programme suivant :

```
Sub Salaire_annuel()

Dim salaire As Single
Dim nom As String

nom = Range("a1")
salaire = InputBox(" " & nom & ": Veuillez entrer votre salaire annuel:", _
"Saisie du salaire de l'utilisateur", 150000)

End Sub
```



Boîte de dialogue *MsgBox*

Il peut être utile de fournir un renseignement à l'utilisateur pendant l'exécution d'un programme. Dans ce cas, l'utilisation de la boîte *MsgBox* est appropriée.

Syntaxe – Afficher une information

```
MsgBox "Message"
MsgBox("Message", Icône, "Titre de la boîte")
```

Arguments

`"Message"`	est le texte qui apparaîtra dans la boîte de dialogue. Est en quelque sorte le message destiné à l'utilisateur
`Icône`	est facultatif et représente un nombre entier. Ce nombre correspond à l'icône choisie dans la boîte de dialogue.
`"Titre de la boîte"`	est lui aussi facultatif, correspond au titre de la boîte de dialogue.

Exemple

```
Sub Investigation()

      MsgBox "Cet employé n'est plus rentable."

End Sub
```

Il est également possible d'obtenir des informations de la part de l'utilisateur avec la boîte MsgBox, lorsqu'il s'agit de répondre à une question par l'affirmative ou la négative.

Syntaxe – Obtenir une information simple

```
Variable = MsgBox("Message", BoutonsEtIcone, "Titre de la boîte")
```

Arguments

`"Message"`	est le texte qui apparaîtra dans la boîte de dialogue. Est en quelque sorte le message destiné à l'utilisateur
`BoutonsEtIcone`	est facultatif et représente un nombre entier. Ce nombre correspond aux boutons et icônes qui apparaîtront dans la boîte de dialogue. Voir tableau plus bas.
`"Titre de la boîte"`	est lui aussi facultatif, correspond au titre de la boîte de dialogue.

Remarques

- La variable « Variable » est de type *Integer*.
- L'argument `BoutonsEtIcone` peut s'entrer de plusieurs façons. Voici un tableau qui les résume :

Constante	*Valeur*	*Description*
Bouton		
`vbOKnly`	0	OK
`vbOKCancel`	1	OK et Annuler
`vbAbortRetryIgnore`	2	Abandonner, Réessayer et Ignorer
`vbYesNoCancel`	3	Oui, Non et Annuler
`vbYesNo`	4	Oui et Non
`vbRetryCancel`	5	Réessayer et Annuler
Symbole ou Icône		
`vbCritical`	16	Message critique
`vbQuestion`	32	Question
`vbExclamation`	48	Stop
`vbInformation`	64	Information
Bouton actif par défaut		
`vbDefaultButton1`	0	Premier bouton
`vbDefaultButton2`	256	Deuxième bouton
`vbDefaultButton3`	512	Troisième Bouton
`vbDefaultButton4`	768	Quatrième bouton

- Il est possible de combiner boutons et icônes en additionnant les valeurs de la colonne du centre. Voir exemple.
- Lorsque l'utilisateur appuie sur une touche (oui, non, annuler, etc.), la commande *MsgBox* retourne une valeur entière. Les valeurs sont résumées dans le tableau suivant.

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

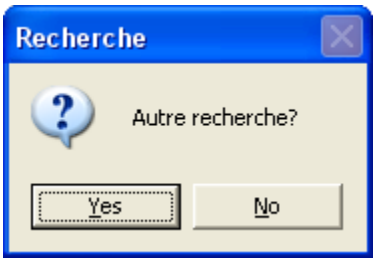
Ce tableau se lit de la façon suivante : si l'utilisateur appuie sur le bouton Oui, alors MsgBox retourne la valeur 6. La variable `Variable` a donc le nombre 6 stocké.

Note : Ces deux tableaux sont tirés du livre Excel 2000 et Visual Basic pour Applications 6

Exemple

Si par exemple nous désirons afficher une boîte de dialogue contenant le symbole « Question », dans laquelle l'utilisateur pourra répondre à la question posée par « Oui » ou « Non », l'argument `BoutonsEtIcone` sera égal à 36 : 4 pour les boutons + 32 pour le symbole. Les trois instructions suivantes sont équivalentes et produisent la même boîte :

- `reponse = MsgBox("Autre recherche?", 36, "Recherche")`
- `reponse = MsgBox("Autre recherche?", 32 + 4, "Recherche")`
- `reponse = MsgBox("Autre recherche?", vbYesNo + vbQuestion, "Recherche")`

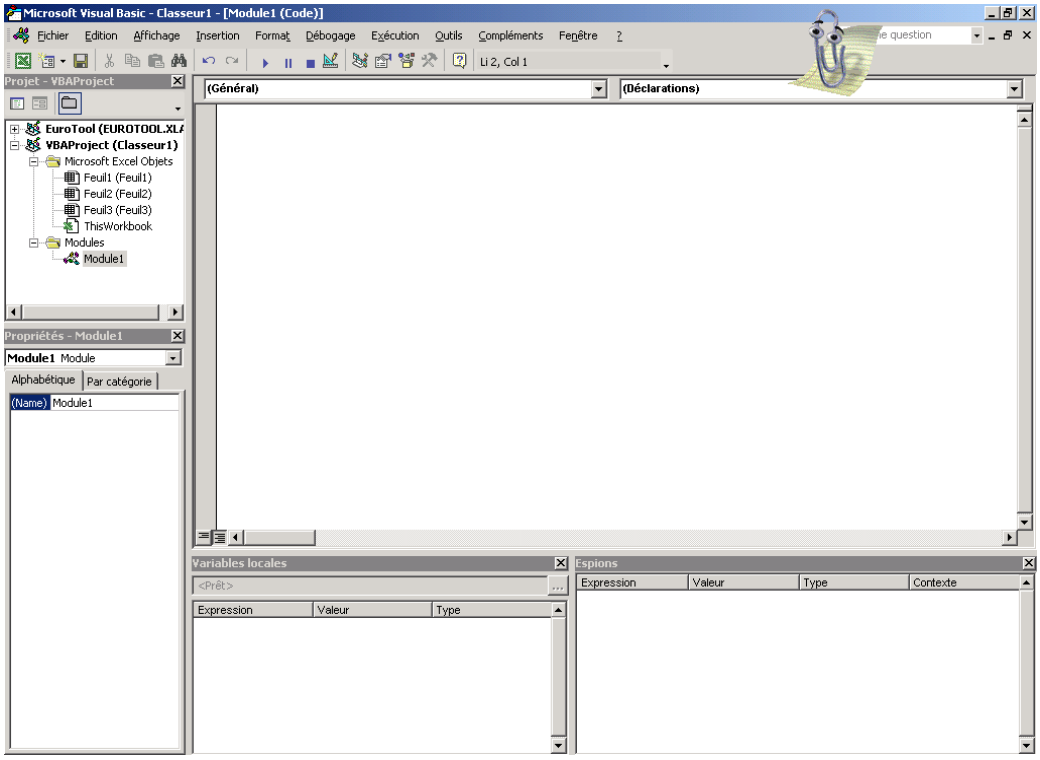


Dans les instructions précédentes, la variable `reponse` contient la réponse de l'utilisateur à la question. Cette réponse est en fait un chiffre; le programme interprétera ensuite ce chiffre et agira en conséquence, selon les désirs du programmeur.

Exemple tiré du livre Excel 2000 et Visual Basic pour Applications 6.

Une vue d'ensemble de l'éditeur Visual Basic

Visual Basic Editor est l'environnement de développement intégré de VBA. Voici un aperçu de Visual Basic Editor :

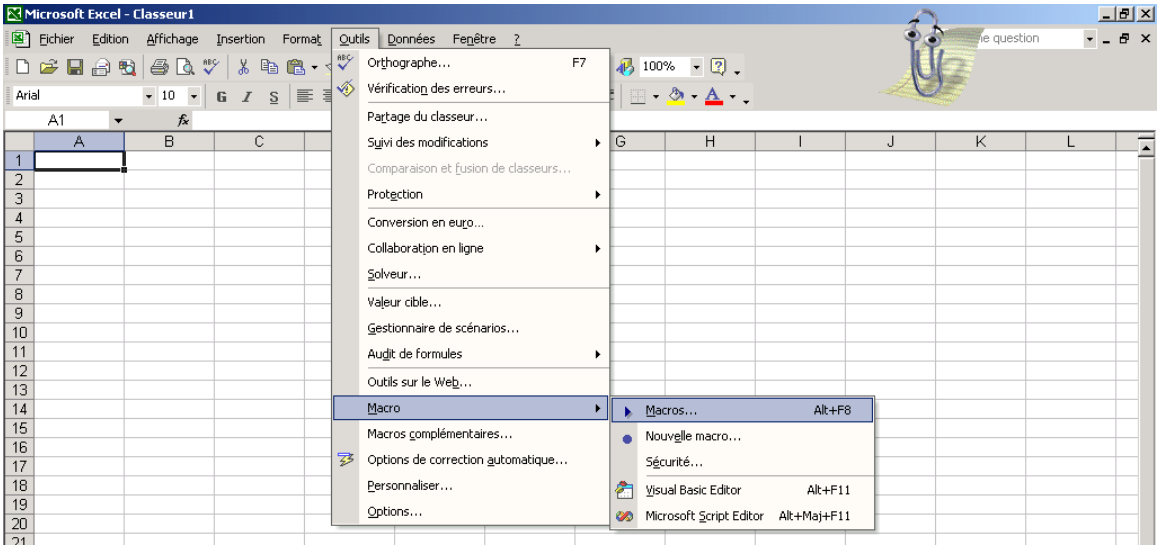


Intéressons-nous à quelques éléments qui pourront vous êtres utiles.

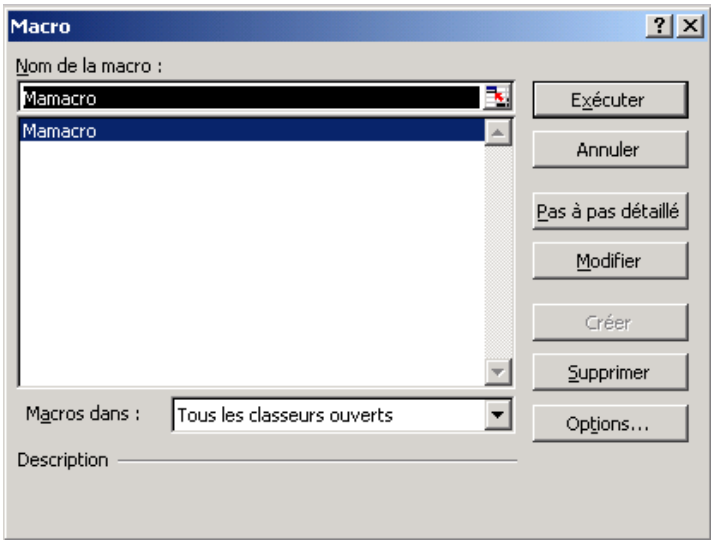
Trucs de débogage

Commande Exécuter

La commande *Exécuter* permet de mettre en marche la macro construite avec VBA. Il existe deux façons de procéder à l'exécution de la macro. La façon la plus rapide consiste à appuyer sur la touche  située dans le haut de la fenêtre de commande. Pour exécuter la macro sans accéder à l'éditeur VBA, on doit choisir Outils/Macro/Macros comme l'illustre la figure suivante :



Voici la boîte de dialogue qui apparaîtra :



Il ne suffit que d'enfoncer le bouton *Exécuter* pour exécuter la macro souhaitée.

Il est également possible d'exécuter une macro à l'aide de la touche F5.

Exécuter Pas à pas

Lorsqu'un programme ne produit pas le résultat escompté, sans générer d'erreurs, le premier test consiste à exécuter la procédure *pas à pas*, afin d'en examiner le déroulement et les conséquences sur le document, instruction après instruction.

Pour exécuter un programme VBA pas à pas, on doit adopter la procédure suivante :

(1) Placer le curseur à la ligne où vous souhaitez démarrer l'exécution pas à pas dans la fenêtre de code de la procédure à tester;
(2) Enfoncer la touche F8 pour exécuter la première ligne du code;
(3) À chaque pression de la touche F8, la ligne de code suivante sera exécutée.

Si vous êtes convaincu de l'exactitude de votre programme jusqu'à une certaine ligne, positionnez votre curseur à cette ligne, appuyez sur CTRL+F8 (appuyez sur CTRL et maintenez enfoncé, puis appuyez sur F8). Visual Basic aura exécuté les lignes avant le curseur et est prêt à exécuter les prochaines lignes, pas-à-pas, avec vous.

Truc : lors de la lecture pas à pas d'une procédure, il peut être intéressant de voir ce que contiennent les variables au fur et à mesure qu'elles se remplissent. Lorsque vous exécutez une procédure en mode pas à pas, vous n'avez qu'à placer le curseur de la souris pour voir ce que contient une variable.

Points d'arrêt

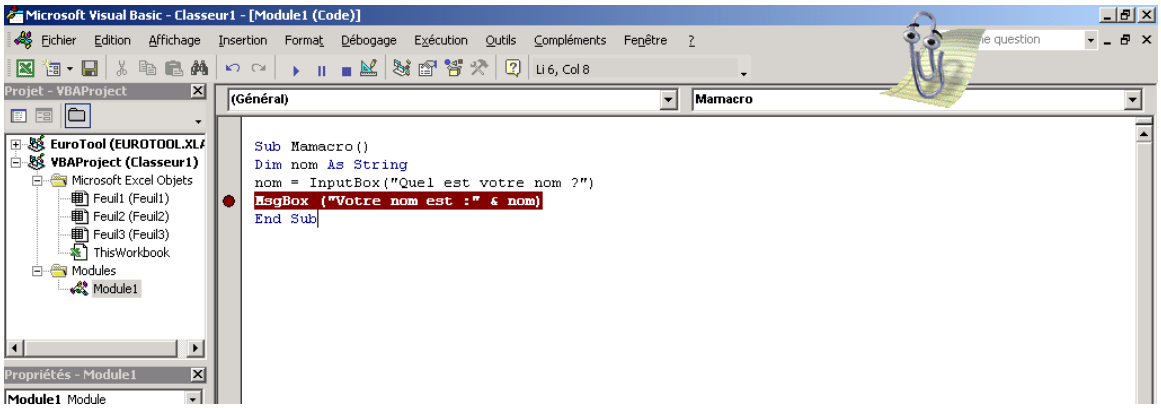
Les points d'arrêt permettent d'interrompe l'exécution d'un programme sur une instruction précise. Cette possibilité est particulièrement intéressante lorsque vous soupçonnez l'origine d'une erreur. Vous pouvez ainsi exécuter normalement toutes les instructions ne posant pas de problèmes et définir un point d'arrêt pour une instruction dont vous n'êtes pas sûr.
La procédure à adopter afin d'insérer un point d'arrêt est simple :

1. Placez le curseur sur l'instruction voulue;
2. Choisir la commande Basculer le point d'arrêt du menu Débogage ou appuyer sur la touche F9;
3. Pour supprimer le point d'arrêt, procéder de la même façon que pour insérer un point d'arrêt.

Autre technique

Il suffit de cliquer en marge de l'instruction à gauche (zone grise). L'instruction en question sera mise en évidence en rouge vin, et un rond rouge vin viendra se placer en marge. Pour enlever le point d'arrêt, il suffit de cliquer dessus à nouveau.

Voici une brève illustration :



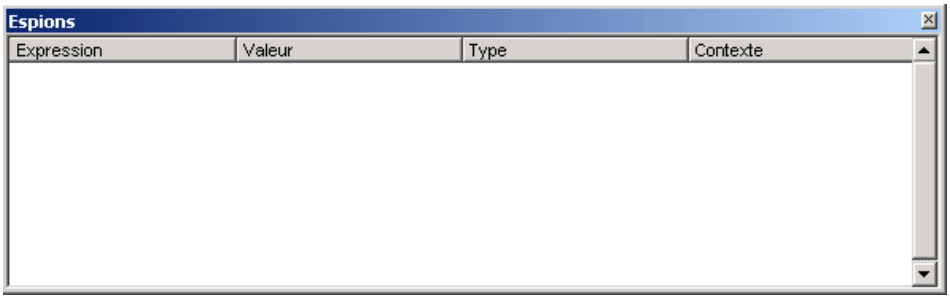
Fenêtre espions

Les espions permettent d'espionner les valeurs de variables ou de toute expression renvoyant une valeur dans un contexte déterminé.

Voici la procédure à adopter pour créer un espion :

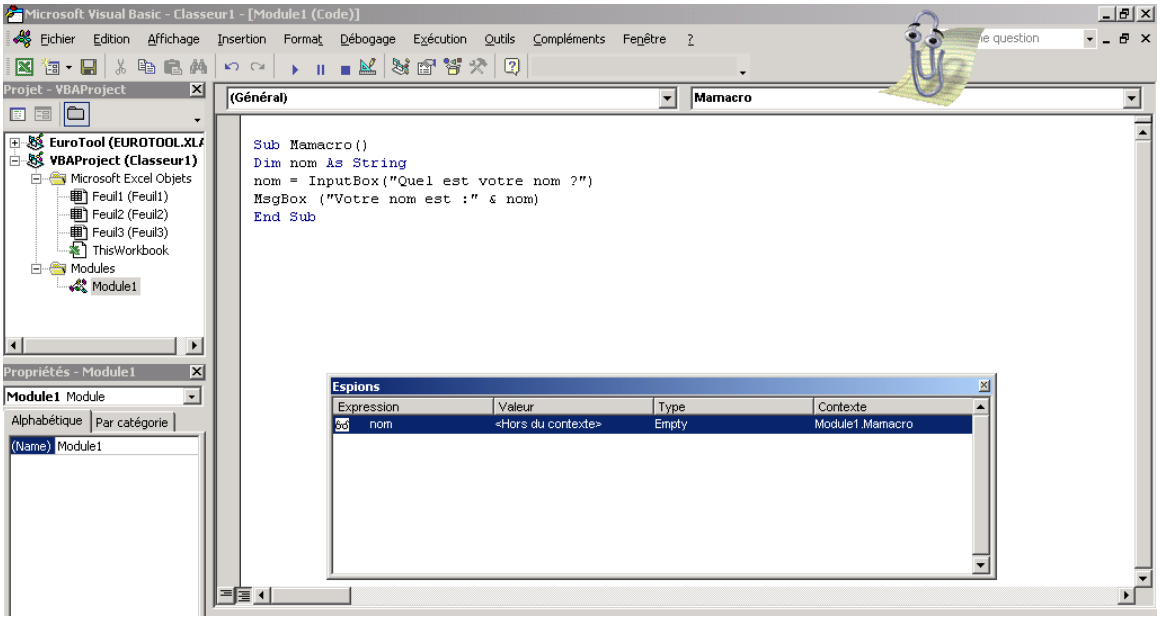
(1) Choisissez la commande *Fenêtre espions* du menu *Affichage;*

La fenêtre suivante apparaîtra :



(2) Mettre en surbrillance la variable à espionner dans la fenêtre de code, cliquer sur cette variable et la glisser dans la fenêtre espions en maintenant enfoncée la touche de gauche de la souris.

Dans l'exemple précédent, on obtiendrait le résultat suivant :



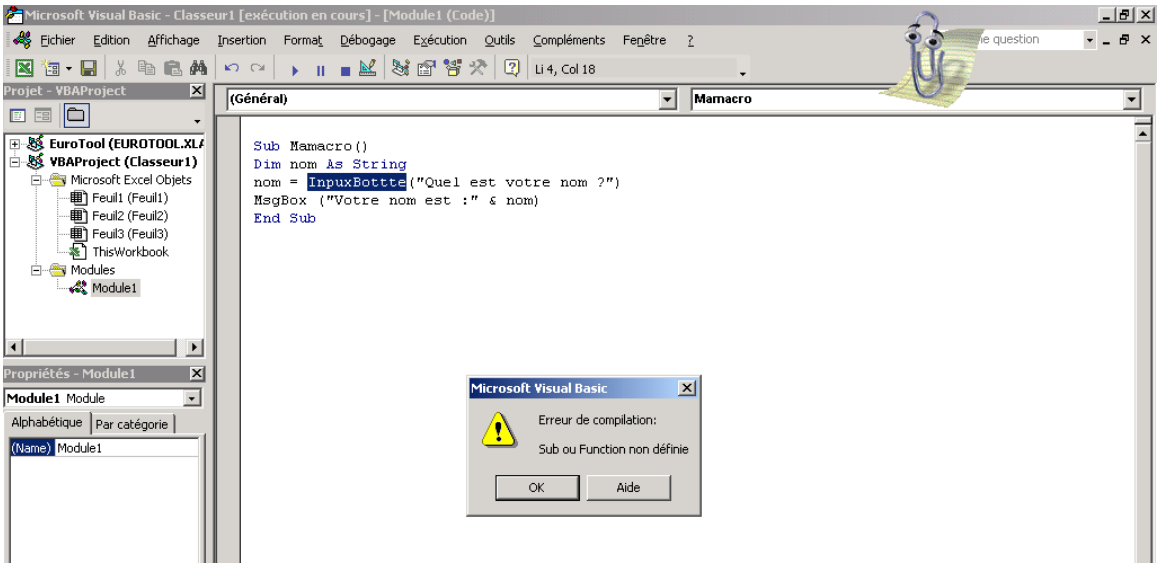
En exécutant le code en mode pas-à-pas ou à l'aide de points d'arrêt, vous serez en mesure de voir le contenu des variables (placées dans la fenêtre espion) se mettre à jour.

Messages d'erreurs

Le débogage consiste donc à régler les erreurs directement liées au code du programme et indépendantes de l'environnement dans lequel s'exécute le programme. Trois principaux types d'erreurs peuvent affecter un programme VBA soit les erreurs de compilation, les erreurs d'exécution et les erreurs logiques.

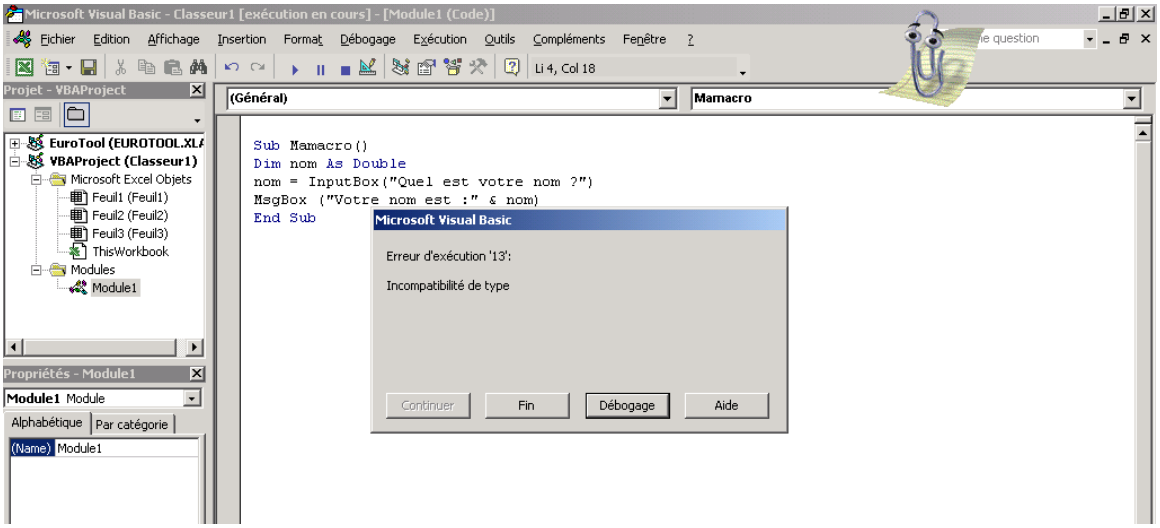
Erreurs de compilation

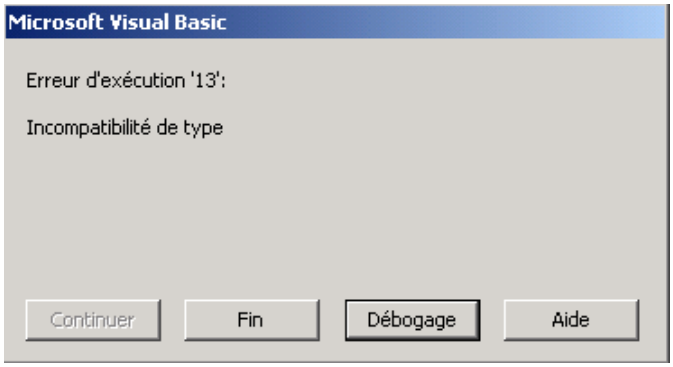
Ce type d'erreur survient lorsque VBA rencontre une instruction qu'il ne reconnaît pas; par exemple, lorsqu'un mot clé contient une faute d'orthographe. On remarque que ce type d'erreur est généralement le plus fréquent. Si on se réfère à l'exemple précédent, on aurait pu obtenir ce message d'erreur :



Erreurs d'exécution

Ce type d'erreur survient après que la compilation du programme a été effectuée avec réussite. Une erreur d'exécution peut par exemple, être liée à l'utilisation de données incompatibles. Dans notre exemple, si on avait déclarer la variable *nom* comme un double, on aurait obtenu le message d'erreur suivant si on avait entré un nom en lettres lors de l'exécution du programme.





Dans la fenêtre ci-dessus le bouton *Fin* permet de terminer l'exécution du programme, le bouton *Débogage* met en surbrillance la ligne de code où VBA détecte une erreur et le bouton *Aide* affiche la rubrique d'aide associé à l'erreur reconnue.

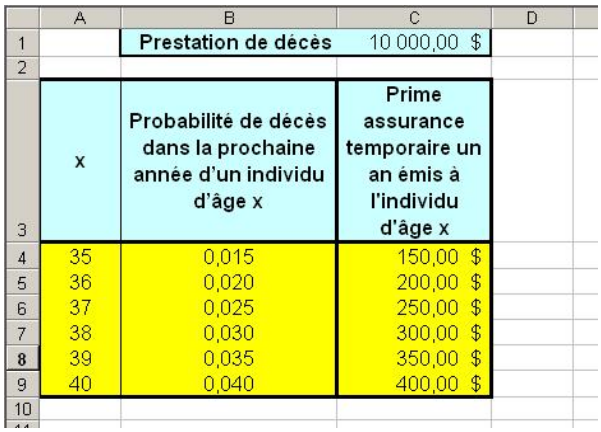
Erreurs logiques

Ce type d'erreur est le plus ardu à corriger. Contrairement aux erreurs de compilation et d'exécution, elles laissent le programme s'exécuter. Le résultat obtenu ne sera pas celui que vous escomptiez. Les erreurs logiques découlent généralement d'une lacune de l'algorithme. Une combinaison de la méthode pas à pas et de l'utilisation des espions est une bonne méthode pour contrer ce type d'erreurs.

Suite de l’exemple – Construction du tableau de primes

La suite du code de l’exemple a été conçue de façon à pouvoir pratiquer le plus de concepts possibles. Il n’est pas nécessairement le plus efficace.

Les lignes qui suivent comprennent tout le code nécessaire pour reproduire l’illustration suivante.



	A	B	C	D
1		Prestation de décès	10 000,00 $	
2				
3	x	Probabilité de décès dans la prochaine année d’un individu d’âge x	Prime assurance temporaire un an émis à l’individu d’âge x	
4	35	0,015	150,00 $	
5	36	0,020	200,00 $	
6	37	0,025	250,00 $	
7	38	0,030	300,00 $	
8	39	0,035	350,00 $	
9	40	0,040	400,00 $	
10				

```
'Avec ce programme, les cellules C4:C9 ne seront pas mises à jour si on change
'la prestation de décès. Il faudra exécuter le programme à chaque fois que la
prestation
'change.
'Cette routine est fonction d'une pretation de décès optionnelle
Sub CreerTableau(Optional Prestation As Double = 10000)

    'Ces deux lignes servent à saisir l'information dans les cellules
    Range("B1").Value = "Prestation de décès"
    Range("C1").Value = Prestation

    'Ces lignes servent à changer l'apparence du contenu des cellules
    With Range("B1:C1")
        .Font.Name = "Arial"
        .Font.Size = 12
        .Interior.ColorIndex = 34
        .Borders(xlEdgeTop).Weight = xlMedium
        .Borders(xlEdgeBottom).Weight = xlMedium
        .Borders(xlEdgeLeft).Weight = xlMedium
        .Borders(xlEdgeRight).Weight = xlMedium
    End With
    Range("B1").Font.Bold = True
    Range("C1").Style = "Currency"

    'Ces lignes servent à modifier la largeur des colonnes
    Columns("B").ColumnWidth = 25
    Columns("C").ColumnWidth = 15

    'Ces lignes servent à entrer le texte
    Range("A3").Value = "x"
    Range("B3").Value = "Probabilité de décès dans la prochaine année d’un individu
d’âge x"
    Range("C3").Value = "Prime assurance temporaire un an émis à l'individu d'âge
x"
```

```
Dim PlagePrimes As Range
Set PlagePrimes = Range("C4:C9")
For i = 0 To 5
    Cells(4 + i, 1) = 35 + i
    Cells(4 + i, 2) = 0.015 + 0.005 * i
    PlagePrimes.Cells(i + 1).Value = Prestation * Cells(4 + i, 2).Value
Next i

'Les prochaines lignes servent à formatter le tableau (sans la bordure)

Dim PlageTitre As Range
Dim PlageContenu As Range

Set PlageTitre = Range("A3:C3")
Set PlageContenu = Range("A4:C9")

PlageTitre.Font.Name = Arial
PlageTitre.Font.Size = 12
PlageTitre.Font.Bold = True
PlageTitre.WrapText = True
PlageTitre.VerticalAlignment = xlVAlignCenter
PlageTitre.HorizontalAlignment = xlHAlignCenter
PlageTitre.Interior.ColorIndex = 34

With PlageContenu
    .Font.Name = Arial
    .Font.Size = 12
    .HorizontalAlignment = xlHAlignCenter
    .Interior.ColorIndex = 6
    .Columns(2).NumberFormat = "0.000"
    .Columns(3).Style = "currency"
End With

'Les prochaines lignes appliquent la bordure
With Range("A3:C9")
    .Borders(xlEdgeTop).Weight = xlThick
    .Borders(xlEdgeBottom).Weight = xlThick
    .Borders(xlEdgeLeft).Weight = xlThick
    .Borders(xlEdgeRight).Weight = xlThick
End With

With Range("A3:B9")
    .Borders(xlEdgeTop).Weight = xlThick
    .Borders(xlEdgeBottom).Weight = xlThick
    .Borders(xlEdgeLeft).Weight = xlThick
    .Borders(xlEdgeRight).Weight = xlThick
End With

With Range("A4:C9")
    .Borders(xlEdgeTop).Weight = xlThick
    .Borders(xlEdgeBottom).Weight = xlThick
    .Borders(xlEdgeLeft).Weight = xlThick
    .Borders(xlEdgeRight).Weight = xlThick
End With

Range("A3:A9").Borders(xlEdgeRight).Weight = xlThin

'Les valeurs xlThin, xlThick, etc. sont des nombres dont la valeur
```

```
    'a déja été stockée en mémoire dans ces constantes systèmes.
End Sub

'Cette routine permet d'appeler la routine CreerTableau avec une nouvelle
'prestation de décès
Sub PRINCIPAL()
    CreerTableau 15000
End Sub
```

Remarques

- Pour que les cellules C4:C9 soient mises à jour dès que l'on modifie la prestation de décès, on peut enlever la ligne
  `PlagePrimes.Cells(i + 1).Value = Prestation * Cells(4 + i, 2).Value`
  et utiliser la propriété *Formula* associé à un objet *Range*.
  Exemple
  `Range("C4").Formula = "=B4*$C$1"`
- Vous pouvez aussi mettre en pratique toutes vos connaissances de Visual Basic et Visual Basic pour Excel en faisant l'exercice actuariel # 5 et en parcourant rapidement les travaux pratiques de l'année 2000.

Références

Bidault, Mikaël. Excel 2000 & Visual Basic pour Applications 6. Campus Press, Simon & Schuster Macmillan. 1999. 534 pages.

Aide en ligne de Microsoft Excel et Visual Basic pour Excel.

Notes de cours, Basic et Visual Basic, Année 2000.