Plan du cours

- VBA, historique
- L'environnement de développement sous Excel
- Les feuilles
- Les contrôles
- La programmation en VBA
- La structure alternative
- La structure répétitive
- Le tableau

Plan du cours

- L'enregistrement
- Le traitement d'erreur
- L’utilisation de DLLs
- Le concept de classe

VBA, historique

- VBA acronyme de Visual Basic for Applications
  - fait partie de la famille Visual Basic (Basic acronyme de Beginners All-purpose Symbolic Instruction Code créé à partir de 1963)
    - Visual Basic (pour Windows) a été lancé en 1991
    - VB est un langage de programmation général permettant de créer “facilement” et “rapidement” des applications Windows

VBA, historique

- VBA
  - a été lancé en 1993 pour remplacer le langage “macro” utilisé jusqu’alors dans Excel
  - permet la création d’applications hébergées dans l’application hôte
  - depuis 1997, les programmes de Office utilisent VBA comme langage de programmation (excepté Outlook) et VBA est vendu pour être intégré dans des applications non Microsoft

VBA, historique

- avantages
  - c'est un langage de programmation complet intégrant tous les concepts usuels
    - types, variables, fonctions, etc.
  - il met à la disposition du programmeur un environnement de développement complet
    - éditeur, débogueur, visualiseur d'objets, etc.
  - le passage de VBA vers VB est aisé

VBA, historique

- avantages
  - il est conçu pour créer de manière “visuelle” des programmes ayant une interface utilisateur graphique
  - il supporte la programmation événementielle
  - il est extensible via l’intégration d’add-ins et/ou de composants Active-X développés en VB ou autre

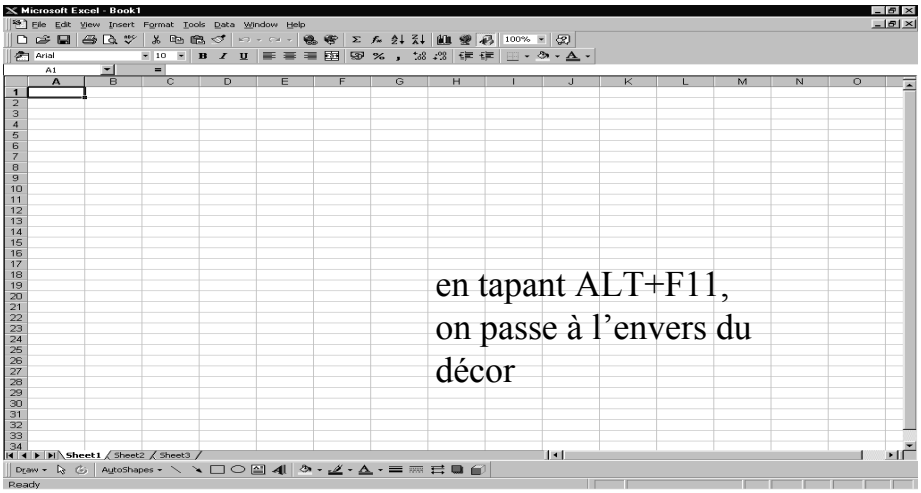
VBA, historique

- avantages
  - langage de programmation puissant
  - présent dans toutes les applications Office

VBA, historique

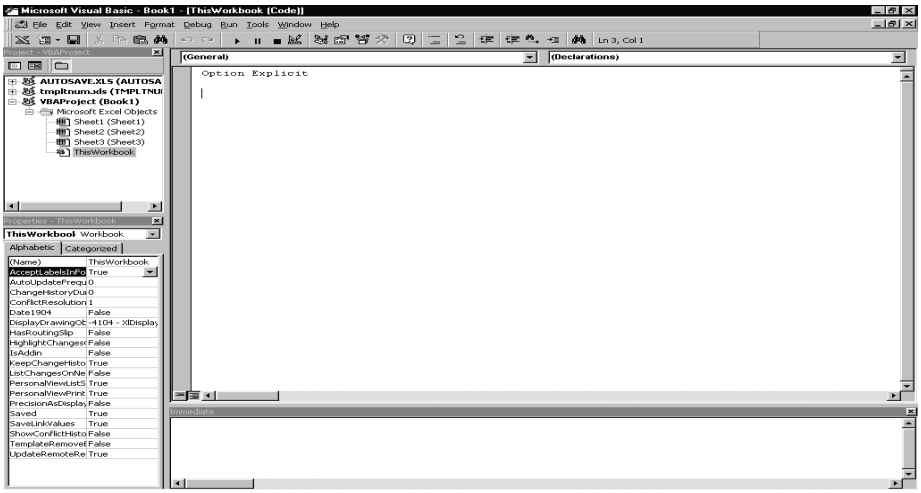
- désavantages
  - c'est un dialecte assez touffu malaisé à appréhender dans sa totalité (plus de 1000 mots-clés)
  - le code est toujours interprété (relativement lent)
  - le langage n'est pas standardisé, il est la propriété de Microsoft qui peut le faire évoluer ou stagner à sa guise !
  - il est n'est pas très facile de diffuser ses programmes sur d'autres machines

L'environnement de développement sous Excel



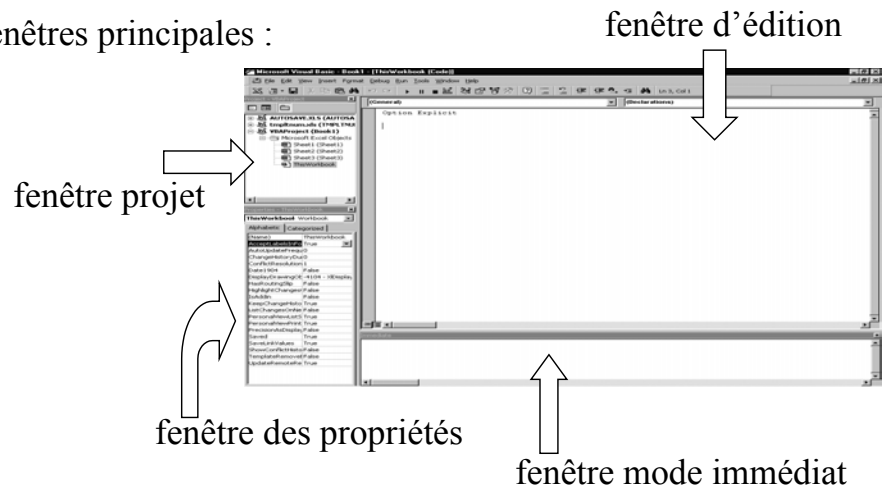
en tapant ALT+F11,
on passe à l'envers du décor

L'environnement de développement sous Excel



L'environnement de développement sous Excel

Les fenêtres principales :



L'environnement de développement sous Excel

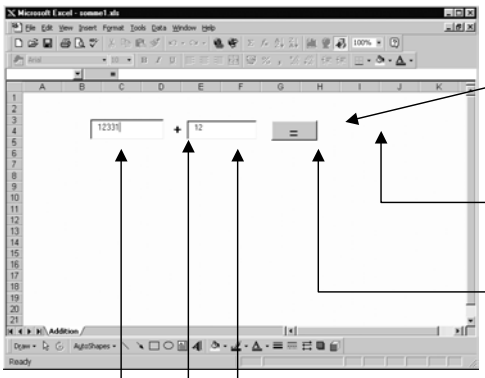
- la réalisation d'une application VBA nécessite en général
  - la saisie par le programmeur d'instructions dans la fenêtre d'édition
  - la création d'un ou de formulaires utilisateur (userform) ou/et l'ajout de contrôles (controls)

L'environnement de développement sous Excel

- lorsque l'application est exécutée, la saisie des données et l'affichage des résultats se fait dans une feuille Excel
- l'utilisateur final ne voit pas l'environnement de développement
- celui-ci ne sert qu'au programmeur à développer et à mettre au point ses programmes VBA

L'environnement de développement sous Excel

Exemple :



contrôle "Command Button"

contrôle invisible "Text Box"

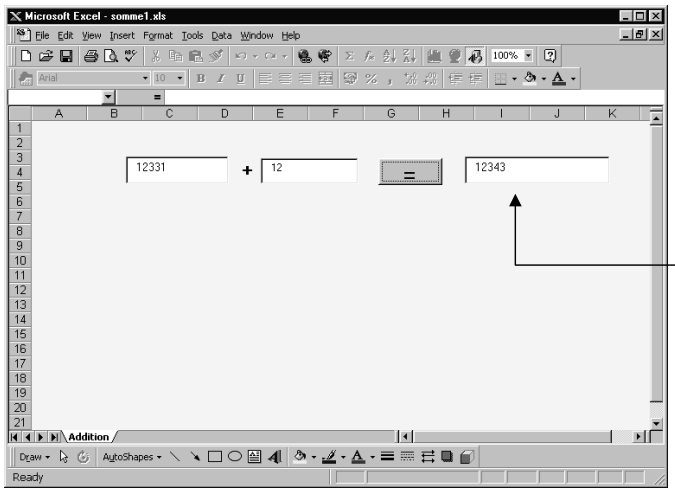
Lorsque l'utilisateur clique sur ce bouton, on obtient le résultat demandé !

contrôle "Text Box"

contrôle "Label"

contrôle "Text Box"

L'environnement de développement sous Excel

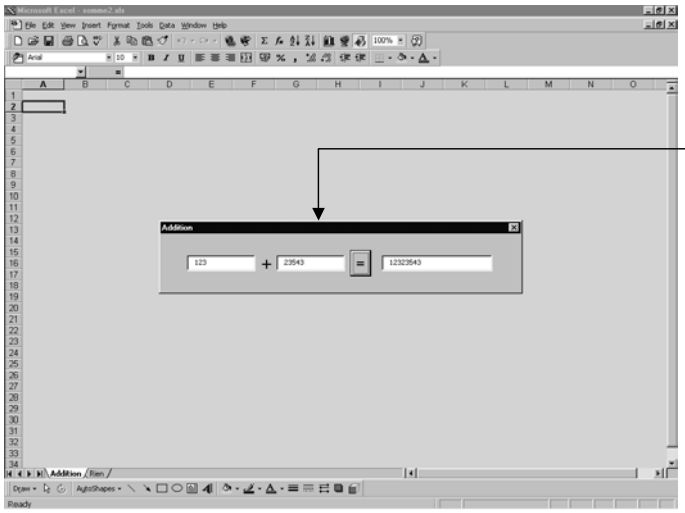


le contrôle invisible "Text Box" est devenu visible !

L'environnement de développement sous Excel

- dans Excel, le programmeur peut placer directement des contrôles dans une feuille de calcul
- mais, il peut aussi les placer dans une/des feuille(s) utilisateur (userform)
- ou faire un mélange des deux

L'environnement de développement sous Excel



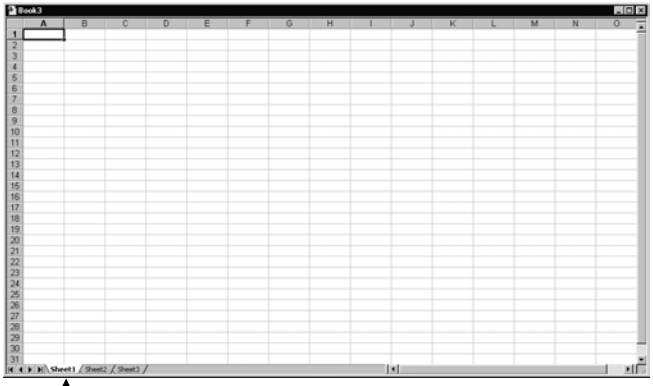
tous les contrôles sont placés dans une feuille de travail utilisateur (userform) appelée "Addition"

Les feuilles

- une feuille
  - est un cadre dans lequel les éléments d'une application VBA peuvent être disposés
  - a des propriétés
  - peut réagir à des événements

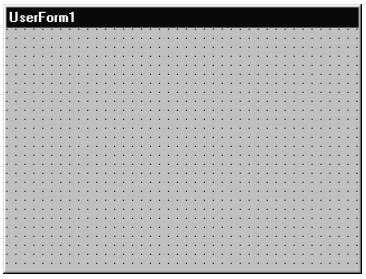
Les feuilles

une feuille de calcul Excel



la feuille visible porte le nom “Sheet1”

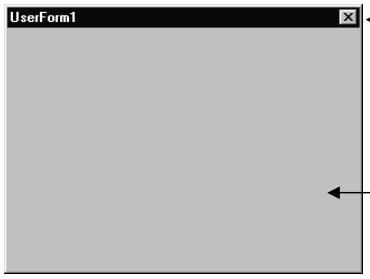
Les feuilles



une feuille utilisateur vierge que le programmeur peut définir à sa guise notamment à l'aide des contrôles

le menu Toolbox visualisant la palette des contrôles usuels s'ouvre automatiquement lorsque le programmeur insère un "userform"

Les feuilles



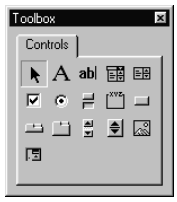
le bouton “fermeture” est inséré automatiquement par Excel

à l'exécution, les points de la grille ont disparu, ils ne servent que comme repères au programmeur

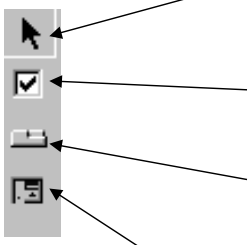
la même feuille utilisateur telle qu’elle se présente à l’utilisateur final

Les contrôles

- Le programmeur peut ajouter des contrôles
  - directement dans une feuille Excel
    - à l'aide de la barre des outils "Control Toolbox"
  - dans un "userform"
    - à l'aide du menu Toolbox



Les contrôles



Select Objects, (sélectionner un objet), ce n'est pas un contrôle, permet de sélectionner les contrôles

CheckBox (case à cocher)

TabStrip (rangée d'onglets), contrôle qui définit une collection de d'onglets

RefEdit, contrôle qui permet une sélection aisée de cellules (plages) dans une feuille de calcul Excel

Les contrôles

Label (étiquette, intitulé), permet de définir une zone texte à afficher

OptionButton (bouton d'option)

MultiPage (rangée de pages), contrôle qui définit une collection de pages, c.-à-d. des containers pour d'autres contrôles

Les contrôles

TextBox (zone de texte), permet de définir une zone de saisie

ToggleButton (bouton bascule, interrupteur)

ScrollBar (défilement), contrôle qui permet d'ajouter une barre de défilement

Les contrôles

ComboBox (zone de liste modifiable)

Frame (cadre), insère un cadre qui permet de regrouper plusieurs contrôles

SpinButton (toupie), contrôle qui permet d'ajouter une toupie

Les contrôles

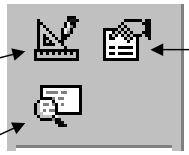
ListBox (zone de liste)

CommandButton (bouton de commande),

Image (image), contrôle qui permet d'ajouter une image

Les contrôles

- la barre des outils “Control Toolbox” présente deux parties en plus des différents contrôles



Design Mode
mode design

View code
voir le code

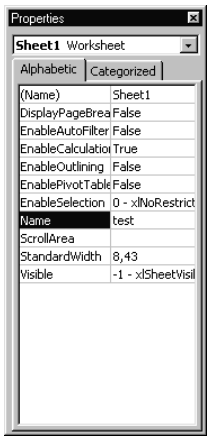
Properties
propriétés

More Controls
accéder à des contrôles
supplémentaires

Les contrôles

- pour placer des contrôles directement dans une feuille Excel, il faut entrer dans le “design mode”
- tant qu’on est en “mode design”, les contrôles ne fonctionnent pas, les actions qui leurs sont associées ne se déclenchent pas !

Les contrôles

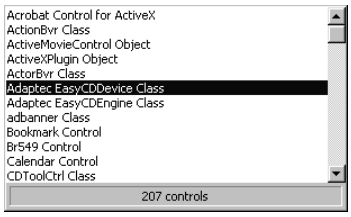


- en cliquant sur l'icône "Properties", la fenêtre des propriétés s'affiche
- le programmeur peut dès lors modifier certaines caractéristiques du contrôle sélectionné

Les contrôles

- l'icône "View Code" amène l'environnement de développement d'Excel en premier plan
- le curseur est positionné dans la fenêtre d'édition

Les contrôles



En cliquant sur “More Controls”, le système affiche la liste de tous les contrôles installés sur la machine du programmeur.

Les contrôles

- pour obtenir la liste des contrôles supplémentaires en travaillant avec un “userform”, il suffit de faire un “clic droit” dans la barre d’outils “Toolbox”

La programmation en VBA

- Ecrire un programme qui effectue une transformation de miles en km.
- L'utilisateur entre le nombre de miles et le programme affiche l'équivalent en kilomètres sachant que 1 mile = 1609 m.

Pour résoudre un problème par des moyens informatiques, il n'y a pas de magie: il faut réaliser un programme qui traite les données qu'on lui fournit pour produire les résultats voulus.

Cela signifie en clair que généralement nous savons quelles sont les données et quels sont les résultats à produire.

Ainsi pour résoudre un problème, il faut faire l'inventaire des données et des résultats.

Dans le cas de notre premier exemple, nous avons comme :

donnée : le nombre de miles

résultat : le nombre équivalent en kilomètres

Ces deux valeurs se trouveront quelque part en mémoire centrale de l'ordinateur. Dans un langage de haut niveau, l'emplacement exact et la représentation interne des valeurs ne nous préoccupent généralement pas. Pour manipuler ces valeurs, nous les désignons par des noms symboliques.

La programmation en VBA

- donnée(s) : identificateur
  le nombre de miles miles

- résultat(s) : identificateur
  l'équivalent en km km

A l'intérieur d'un programme, toute valeur est donc généralement représentée par un identificateur. Le nom de l'identificateur est laissé au choix du programmeur.

Mais celui-ci a naturellement intérêt à choisir un nom proche de la désignation de la valeur que l'identificateur est censé représenter.

Les règles de construction d'un identificateur diffèrent d'un langage de programmation à un autre.

En VBA, le premier caractère doit être une lettre de A à Z. Il n'est pas permis d'utiliser un espace et les caractères . ! @ & $ #. La taille ne peut excéder 255 caractères.

Les lettres majuscules ou minuscules peuvent être utilisées indifféremment ce qui n'est pas le cas du langage C par exemple.

Ainsi, en VBA :

miLEs miles MILES

ne désignent qu'un seul identificateur, alors qu'en C, ces trois noms sont trois identificateurs différents.

La programmation en VBA

- Attention :
  - un identificateur créé par le programmeur ne devrait pas porter le même nom qu'un objet prédéfini en VBA
- Problème :
  - la liste est très longue et peut changer
- Solution : se reporter à l'aide en ligne

IsArray, Array, Option Base, Dim, Private, Public, ReDim, Static, LBound, UBound, Erase, ReDim, Collection, Add, Remove, Item, #Const, #If, Then, #Else, GoSub, Return, GoTo, On Error, On...GoSub, On...GoTo, DoEvents, End, Exit, Stop, Do, Loop, For, Next, Each, While...Wend, With, Choose, If, Else, Select Case, Switch, Call, Function, Property Get, Property Let, Property Set, Sub, Chr, Format, LCase, UCase, DateSerial, DateValue, Hex, Oct, Format, Str, CBool, CByte, CCur, CDate, CDbl, CDec, CInt, CLng, CSng, CStr, CVar, CVErr, Fix, Int, Day, Month, Weekday, Year, Hour, Minute, Second, Asc, Val, TimeSerial, TimeValue, Boolean, Byte, Currency, Date, Double, Integer, Long, Object, Single, String, Variant, IsArray, IsDate, IsEmpty, IsError, IsMissing, IsNull, IsNumeric, IsObject, Now, Time, DateAdd, DateDiff, DatePart, Timer, ChDir, ChDrive, FileCopy, MkDir, RmDir, Name, CurDir, FileDateTime, GetAttr, FileLen, Dir, SetAttr, Clear, Error, Raise, Error, Err, CVErr, On Error, Resume, IsError, DDB, SLN, SYD, FV, Rate, IRR, MIRR, NPer, IPmt, Pmt, PPmt, NPV, PV, Open, Close, Reset, Format, Print, Print #, Spc, Tab, Width #, FileCopy, EOF, FileAttr, FileDateTime, FileLen, FreeFile, GetAttr, Loc, LOF, Seek, Dir, Kill, Lock, Unlock, Name, Get, Input, Input #, Line Input #, FileLen, FileAttr, GetAttr, SetAttr, Seek, Print #, Put, Write #, Atn, Cos, Sin, Tan, Exp, Log, Sqr, Randomize, Rnd, Abs, Sgn, Fix, Int, DoEvents, AppActivate, Shell, SendKeys, Beep, Environ, Command, MacID, MacScript, CreateObject, GetObject, QBColor, RGB, DeleteSetting, GetSetting, GetAllSettings, SaveSetting, etc.

La programmation en VBA

- Le programmeur doit spécifier le *type*, c'est-à-dire la nature de l'information représentée par un identificateur.
- en VBA, il existe de multiples types prédéfinis :
  - *integer*
  - *double*
  - *string*
  - *etc.*

Une fois que nous avons fixé un identificateur pour une valeur dans la mémoire de l'ordinateur, il nous reste à indiquer quel est le genre, le type de la valeur.

N'oublions pas, en effet, que la mémoire centrale d'un ordinateur se compose de mots mémoire identiques, quelque soit la nature de l'information que l'on y stocke. Donc à la seule vue du mot mémoire, l'ordinateur n'est pas à même de décider à quel type d'information il est confronté.

C'est au programmeur que revient la tâche d'indiquer la nature ou le type d'une valeur. Ainsi, pour tout identificateur qu'il définit, le programmeur doit indiquer au compilateur le type de la valeur représentée par cet identificateur.

Cela permet au compilateur de réserver assez d'espace mémoire pour la représentation de la valeur dans la mémoire. De plus, vu que le compilateur connaît alors le type de l'identificateur, il peut effectuer des contrôles stricts par la suite et détecter des anomalies où le programmeur tente par exemple de "comparer des pommes et des poires" !

La programmation en VBA

- pour travailler avec des nombres, le programmeur a le choix :
  - byte, integer, long,
  - single, double,
  - currency, decimal

Extrait de l'aide en ligne

Data type	Storage size	Range
Byte	1 byte	0 to 255
Boolean	2 bytes	True or False
Integer	2 bytes	-32,768 to 32,767
Long	4 bytes	-2,147,483,648 to 2,147,483,647
Single	4 bytes	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
Double	8 bytes	-1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	14 bytes	+/-79,228,162,514,264,337,593,543,950,335 with no decimal point; +/-7.9228162514264337593543950335 with 28 places to the right of the decimal;

smallest non-zero number is +/-0.0000000000000000000000000001

La programmation en VBA

- d'autres types existent :
  - boolean, date, object, string, variant
  - le concept de tableau (array)
- et le programmeur peut créer ses propres types

Date	8 bytes	January 1, 100 to December 31, 9999
Object	4 bytes	Any Object reference
String (variable-length)		10 bytes + string length (0 to approximately 2 billion)
String (fixed-length)		Length of string (1 to approximately 65,400)
Variant (with numbers)		16 bytes, Any numeric value up to the range of a Double
Variant (with characters)		22 bytes + string length (Same range as for variable-length String)
User-defined (using Type)	Number required by elements	The range of each element is the same as the range of its data type.

Note Arrays of any data type require 20 bytes of memory plus 4 bytes for each array dimension plus the number of bytes occupied by the data itself. The memory occupied by the data can be calculated by multiplying the number of data elements by the size of each element. For example, the data in a single-dimension array consisting of 4 Integer data elements of 2 bytes each occupies 8 bytes. The 8 bytes required for the data plus the 24 bytes of overhead brings the total memory requirement for the array to 32 bytes.

A Variant containing an array requires 12 bytes more than the array alone.

Extrait de l'aide en ligne

La programmation en VBA

Une variable :

- un identificateur
- un type
- une valeur

En programmation impérative (C, Cobol...), les programmes arrivent à produire les résultats désirés en manipulant des variables.

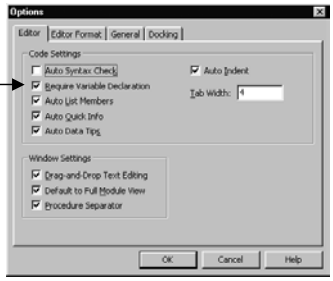
Une variable est un triplet (identificateur, type, valeur). En clair, une variable est désignée par un nom ou identificateur. D'après ce qui précède, un type est associé à l'identificateur en question. Comme cet identificateur désigne des mots dans la mémoire centrale de l'ordinateur, une variable a toujours une valeur.

La valeur est obtenue en interprétant les mots mémoire désignés par l'identificateur de la manière indiqué par le type de la variable.

Cette valeur peut être indéfinie si le programmeur utilise une variable sans que celle-ci n'ait été initialisée. En VBA, une variable numérique est initialisée par défaut à 0, alors qu'une variable de type chaîne de caractères prend comme valeur la chaîne vide "".

La programmation en VBA

- une variable est déclarée explicitement à l'aide du mot réservé DIM
- pour interdire la déclaration implicite de variables
  - il faut placer la directive OPTION EXPLICIT dans chaque programme
  - ou cocher cette case dans le menu Tools/Options

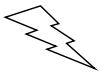


Toute variable devrait être déclarée avant sa première utilisation. Pour des raisons historiques, dans VBA et la plupart des dialectes Basic, une variable peut être utilisée sans dire préalablement au système quel type de données elle est censée contenir. Une telle variable a alors le type "Variant". C'est une pratique à rejeter pour des raisons de performances, d'occupation d'espace mémoire, de sécurité, etc.

Malheureusement, par défaut, VBA fonctionne de cette façon. Il est donc vivement conseillé de désactiver ce comportement en plaçant la directive OPTION EXPLICIT manuellement dans chacun de ses programmes, ou de manière plus pratique, en cochant la case "Require Variable Declaration" dans le menu Tools/Options de l'environnement de programmation.

La programmation en VBA

- Exemples de déclarations de variables :
  - DIM total AS Integer
  - DIM nbre AS Integer, delta AS Double
- Attention : ne pas écrire
  - DIM nbre1, nbre2 AS Integer



nbre1 est du type Variant
nbre2 est du type Integer

- mais :
  - DIM nbre1 AS Integer, nbre2 AS Integer

Une déclaration de variable(s) débute donc par le mot réservé DIM. Deux déclarations de variable dans une instruction DIM sont séparées par le symbole ",".

Dans notre exemple, nous écrirons : Dim miles As Double, km As Double

La programmation en VBA

- l'affectation est l'instruction fondamentale dans les langages impératifs
- l'affectation permet de changer la valeur d'une variable
- il faut que le type de l'expression et le type de la variable soient compatibles !
- format simplifié :

  [LET] variable "=" expression

Dans les langages de programmation impératifs, l'instruction d'affectation est l'instruction fondamentale permettant au programmeur de donner aux variables les valeurs attendues.

L'affectation permet de donner explicitement une valeur à une variable. Lors de l'exécution de cette instruction, l'expression placée à droite du symbole "=" est évaluée. Après évaluation de cette expression, la valeur résultante est transférée dans les cellules mémoire désignées par l'identificateur de variable placé à gauche du symbole d'affectation "=".

Remarque très importante : la valeur qu'avait la variable à gauche du symbole "=" juste avant l'exécution de l'affectation est irrémédiablement perdue !

De ce fait, l'ordre dans lequel les instructions sont placées par le programmeur, est primordial.

Remarque : Dans un format général d'une instruction, une expression placée entre « { } » signifie que l'expression en question peut être répétée de 0 à autant de fois qu'on le désire. Une expression entre « [ ] » est facultative, c'est-à-dire qu'elle est présente une fois ou pas du tout. Le méta-symbole « | » indique qu'on a le choix entre l'expression à sa gauche et à sa droite.

La programmation en VBA

- dans notre exemple :

  km = 1.609 * miles

- la variable *miles*, désignant le nombre de miles à convertir en km est multipliée par 1.609. Cette valeur est affectée ensuite à la variable *km*.

Le programmeur peut définir des expressions très complexes. Voici quelques opérateurs dans leur ordre de priorité décroissant:

^	exponentiation	2^3=8
-	moins unaire	-3
* /	multiplication, division	3/2=1.5
\	division entière	3\2 = 1
Mod	modulo	7 Mod 4 =3
+ -	addition, soustraction	
&	concaténation d'expressions	

En utilisant \ et MOD, les arguments non entiers sont arrondis vers l'entier le plus proche.

Ces opérateurs peuvent être combinés à volonté. Le programmeur doit néanmoins savoir qu'une expression est évaluée de la gauche vers la droite, l'ordre de priorité indiqué prévaut et que seul un placement de parenthèses peut modifier l'ordre d'évaluation à l'intérieur d'une expression.

Il faut évidemment veiller à ce que les types manipulés dans une expression soient compatibles. VBA effectue des conversions de types dans une expression si les opérandes sont de type "nombre" différents.

Attention : le résultat de l'opérateur "&" est un "string"

La programmation en VBA

- il existe deux boîtes de dialogue prédéfinies en VBA
  - InputBox
  - MsgBox
- elles permettent respectivement
  - la saisie d'une valeur au clavier
  - l'affichage d'un message

VBA offre de nombreuses possibilités à travers les contrôles pour saisir et afficher des données.

Deux boîtes de dialogue prédéfinies existent néanmoins pour simplifier la vie du programmeur et pour donner un aspect usuel aux dialogues standards :

InputBox et MsgBox.

La programmation en VBA

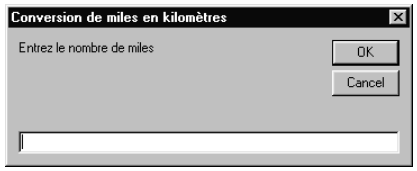
- InputBox affiche une boîte de dialogue, attend que l'utilisateur entre du texte ou clique sur un bouton et retourne un "string" renfermant le texte entré
- Format simplifié :
  - InputBox(prompt [,title] [,default])

Extrait de l'aide en ligne

prompt	Required. String expression displayed as the message in the dialog box. The maximum length of prompt is approximately 1024 characters, depending on the width of the characters used. If prompt consists of more than one line, you can separate the lines using a carriage return character (Chr(13)), a linefeed character (Chr(10)), or carriage return–linefeed character combination (Chr(13) & Chr(10)) between each line.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit title, the application name is placed in the title bar.
default	Optional. String expression displayed in the text box as the default response if no other input is provided. If you omit default, the text box is displayed empty.

La programmation en VBA

- Pour notre exemple, on écrira :
  - miles = InputBox("Entrez le nombre de miles",
    "Conversion de miles en kilomètres")



La programmation en VBA

- MsgBox affiche une boîte de dialogue, attend que l'utilisateur clique sur un bouton de la boîte et renvoie un "Integer" indiquant quel bouton a été cliqué
- Format simplifié :
  - MsgBox(prompt [,buttons] [,title])

Extrait de l'aide en ligne

prompt	Required. String expression displayed as the message in the dialog box. The maximum length of prompt is approximately 1024 characters, depending on the width of the characters used. If prompt consists of more than one line, you can separate the lines using a carriage return character (Chr(13)), a linefeed character (Chr(10)), or carriage return – linefeed character combination (Chr(13) & Chr(10)) between each line.
buttons	Optional. Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If omitted, the default value for buttons is 0.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit title, the application name is placed in the title bar.

La programmation en VBA

- Pour notre exemple, on écrira :
  - res = MsgBox(miles & " miles équivalent à " & km & " kilomètres", _
    vbOKOnly, _
    "Résultat de la conversion")

Le texte affiché à l'intérieur de la boîte résulte de la concaténation des expressions

miles & " miles équivalent à " & km & " kilomètres"

en se servant de l'opérateur "&". Le contenu de la variable *miles* vaut 10 et celui de *km* vaut 16.09. Ces valeurs de type *double* sont transformées en chaînes de caractères et concaténées aux constantes littérales placées entre ".

vbOKOnly est une constante prédéfinie dans VBA. Il dénote un type de boîte où le programmeur n'a que la possibilité de cliquer sur OK.

La valeur entière retournée par *MsgBox* identifie le bouton cliqué par l'utilisateur.

vbOK	1	OK
vbCancel	2	Cancel
vbAbort	3	Abort
vbRetry	4	Retry
vbIgnore	5	Ignore
vbYes	6	Yes
vbNo	7	No

La programmation en VBA

- VBA préfinit d'autres boîtes de dialogue standard
  - OK/Cancel
  - Yes/No
  - Abort/Retry/Ignore
  - etc.

The first group of values (0–5) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512) determines which button is the default; and the fourth group (0, 4096) determines the modality of the message box. When adding numbers to create a final value for the buttons argument, use only one number from each group.

vbOKOnly	0	Display OK button only.
VbOKCancel	1	Display OK and Cancel buttons.
VbAbortRetryIgnore	2	Display Abort, Retry, and Ignore buttons.
VbYesNoCancel	3	Display Yes, No, and Cancel buttons.
VbYesNo	4	Display Yes and No buttons.
VbRetryCancel	5	Display Retry and Cancel buttons.
VbCritical	16	Display Critical Message icon.
VbQuestion	32	Display Warning Query icon.
VbExclamation	48	Display Warning Message icon.
VbInformation	64	Display Information Message icon.
VbDefaultButton1	0	First button is default.
VbDefaultButton2	256	Second button is default.
VbDefaultButton3	512	Third button is default.
VbDefaultButton4	768	Fourth button is default.
VbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
VbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.

Extrait de l'aide en ligne

La programmation en VBA

- nous disposons maintenant de tous les ingrédients de base pour réaliser notre premier exemple
- il existe au moins deux façons de procéder
- il suffit par exemple de créer un bouton de commande dans une feuille Excel et lui associer l'action de conversion demandée

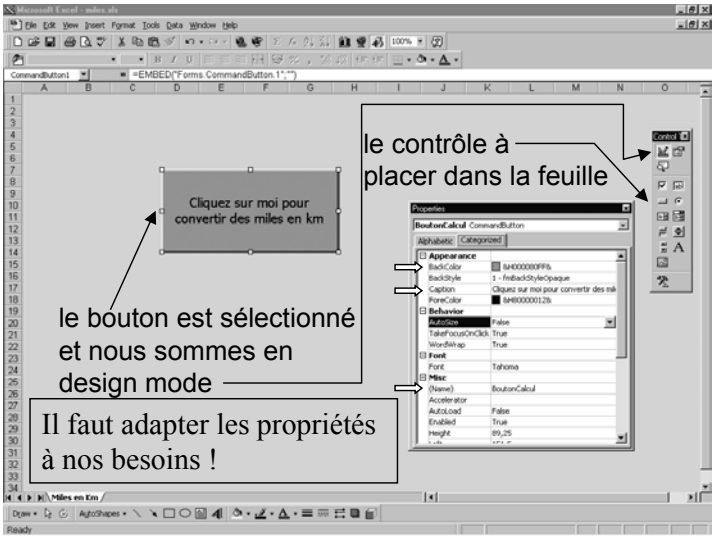
Nous avions déjà vu qu'il existe en principe deux façons de programmer en VBA :

• placer des contrôles directement dans une feuille de calcul Excel

• placer les contrôles dans un "userform" et activer le userform dans sa feuille Excel.

Dans ce premier exemple, nous adoptons la première technique.

La programmation en VBA



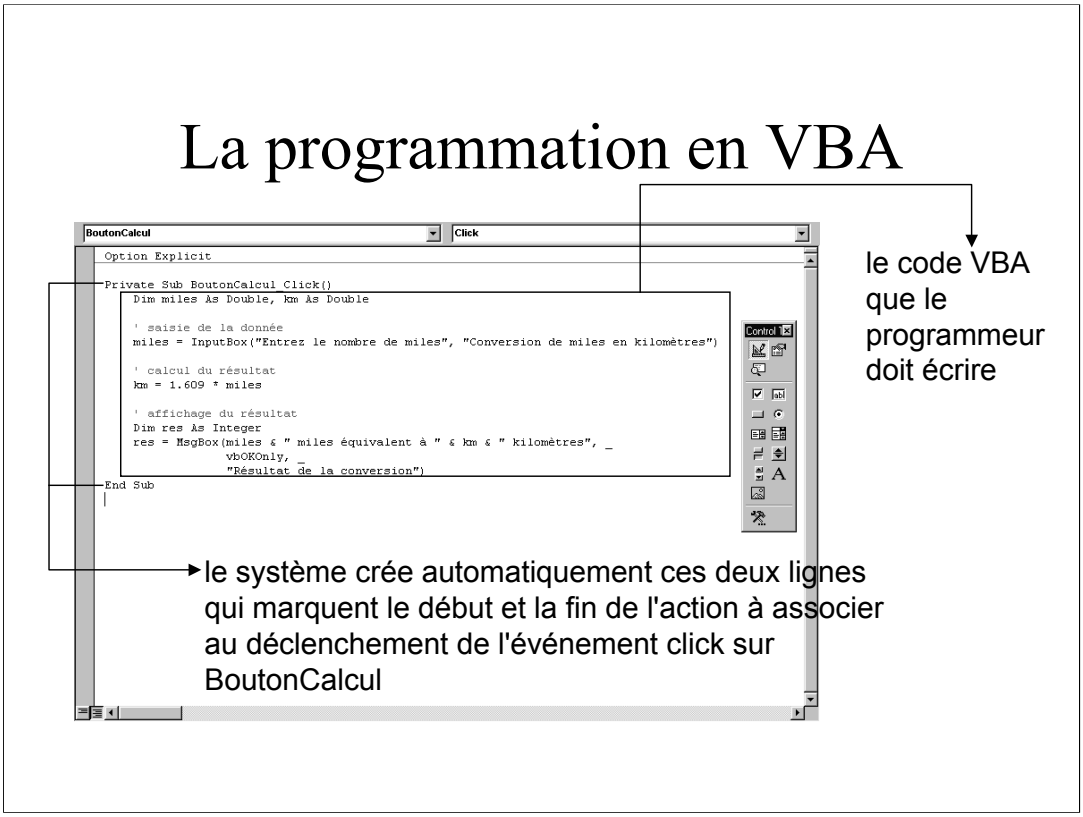
Étapes à suivre :

• placer un contrôle “bouton de commande” sur une feuille Excel vide. Sur l’image, la case à cocher “Gridlines” de Tools/Options dans l’onglet “View” a été désactivé et les cases ont été formatés avec une couleur saumon.

• agrandir le bouton à sa convenance

• changer certaines propriétés du bouton de commande à travers la fenêtre des propriétés. Sur le transparent, elles sont indiquées par des flèches rouges. Pour cet exemple, il s’agit de *Backcolor*, *Caption* et *Name*. *Backcolor* change la couleur du bouton; *Caption* contient le texte qui sera placé dans le bouton et *Name* permet de changer le nom symbolique par défaut du bouton. Il s’agit de bien choisir le nouveau nom car tout changement ultérieur s’avère laborieux ! L’identificateur choisi est “BoutonCalcul”.

• double-cliquer le bouton nous amène dans l’éditeur de l’environnement de développement. Maintenant, il s’agit d’écrire le code associé à l’événement “Click” sur le bouton de commande. C’est ici qu’il faut placer les instructions que l’environnement doit exécuter si par la suite l’utilisateur final clique sur ce bouton.



Une fois dans l'éditeur, le curseur est positionné à l'intérieur d'une procédure que le système a nommé *BoutonCalcul_Click*(). Le nom est la concaténation de l'identificateur que nous avons choisi comme nom pour le contrôle "Bouton de commande" et de l'événement associé à l'action, ici *click*.

En VBA, une procédure est appelée *sub* (subroutine, sous-routine).

Le mot *private* (ajouté automatiquement par le système) figurant avant *sub* signifie que cette procédure n'est visibile est accessible que dans la feuille de calcul Excel qui contient le bouton.

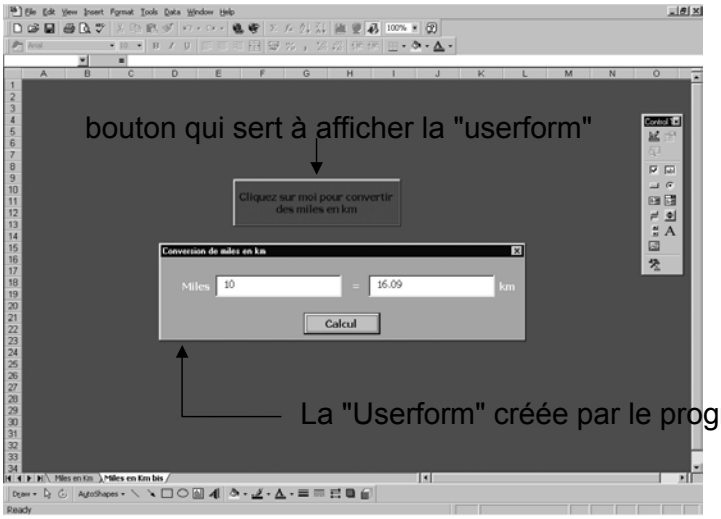
Deux remarques concernant l'écriture du code :

• le symbole ' est le début d'un commentaire, c'est-à-dired'un texte qui est ignoré par l'interprète VBA, mais qui sert à documenter le code écrit.
Il est essentiel de bien documenter ses programmes !

• une ligne trop longue peut être coupée. Il suffit de terminer la ligne qui doit être coupée par le symbole _ et de continuer à la ligne plus bas.

Il ne reste qu'à tester le programme : on peut passer par exemple dans la feuille Excel, s'assurer qu'on est plus en mode design et cliquer sur le bouton qu'on vient de créer!

La programmation en VBA

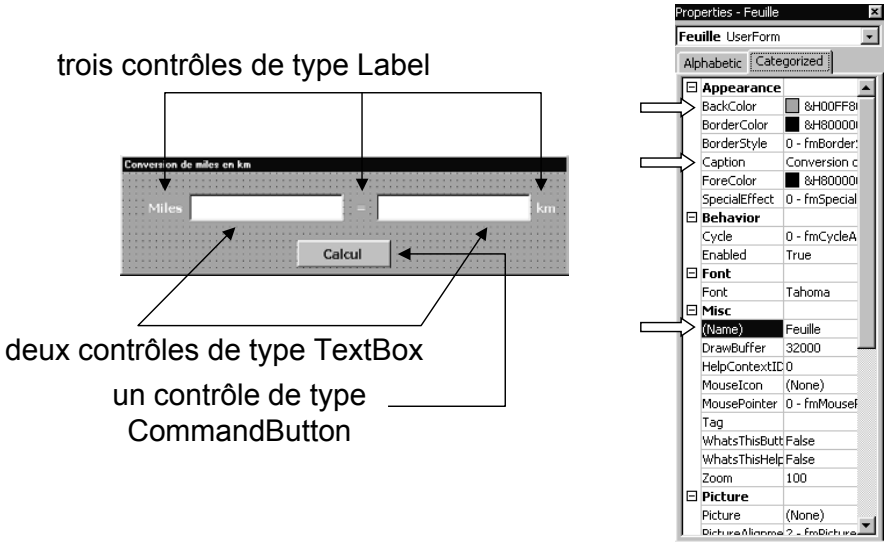


Le programme précédent peut être réalisé avec une interface différente en utilisant un "userform"

Une même solution de problème peut être implémentée par des programmes différents, notamment au niveau de l'interface utilisateur.

Le programme qui convertit des miles en km est réalisé différemment en utilisant une feuille utilisateur "userform" définie dans l'environnement de programmation.

La programmation en VBA



Dans l'environnement de programmation, une feuille utilisateur est inséré dans le projet à travers le menu "Insert/User Form".

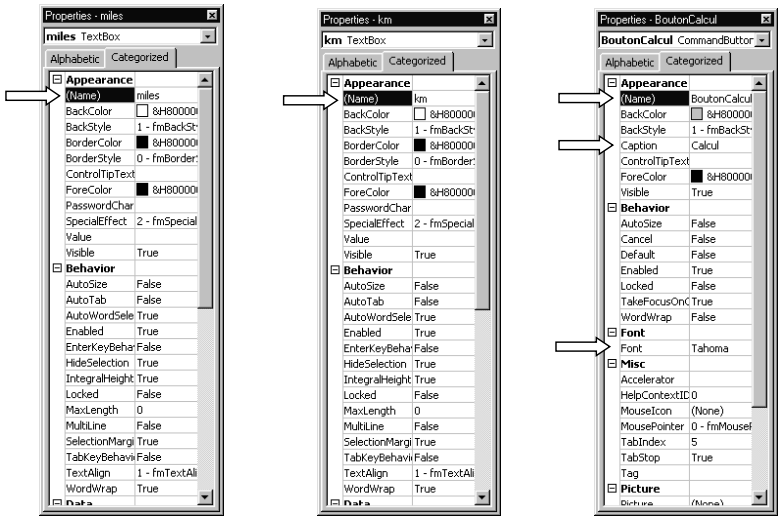
Des propriétés de la feuille ainsi créée peuvent être modifiées par le programmeur. Dans l'exemple ci-dessus, les propriétés modifiées sont marquées par une flèche : il s'agit de la couleur de fond, du titre (*Conversion*…) et du nom (*feuille*) de la userform telle qu'elle est connue par Excel.

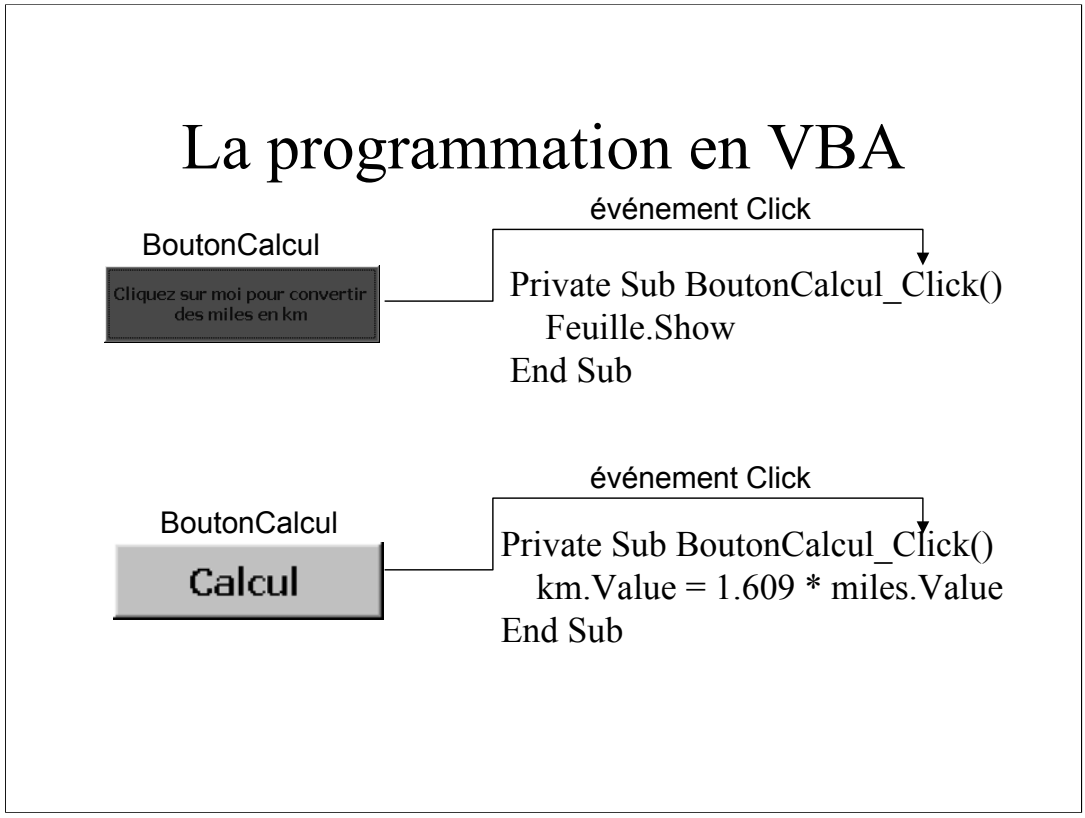
Les contrôles sont ensuite placés dans le userform.

De nouveau nous devons changer certaines propriétés pour les adapter à nous besoins.

Il est essentiel de changer le nom par défaut des contrôles que nous manipulerons plus tard dans les instructions et ceci avant de commencer à programmer les actions associées à ces contrôles.

La programmation en VBA





Après avoir défini le "userform" *Feuille*, nous définissons un bouton de commande *BoutonCalcul* dans la feuille de travail Excel. Son but est d'activer le userform *Feuille* lorsque l'utiliteur clique sur le bouton. Au moment où Excel charge la feuille de travail (worksheet) en mémoire, celle-ci contient le userform *Feuille* que nous veons de créer, mais il est caché. Il suffit de la rendre visible. C'est le travail de la procédure *BoutonCalcul_Click*() associé à ce bouton à travers l'instruction *Show* appliqué à *Feuille* : *Feuille.Show*

A l'exécution, lorsque le userform est actif, l'utilisateur peut entrer des valeurs dans les zones d'édition, dans notre cas *miles*.

En cliquant sur le bouton "Calcul", l'action *BoutonCalcul_Click*() propre à ce bouton est déclenchée, c'est-à-dire l'instruction

km.Value = 1.609 * miles.Value

Elle signifie que la valeur associée au contrôle *km* devient la valeur associée au contrôle *miles* multipliée par 1.609.

Value est une des propriétés du contrôle *TextBox*.

La structure alternative

- la réalisation d'actions n'est pas toujours un processus séquentiel
- beaucoup de situations exigent des traitements spécifiques
- d'où la nécessité d'une structure de contrôle alternative permettant d'introduire la notion de choix

Les exemples réalisés jusqu'à présent avaient la caractéristique que *toutes* les instructions de chacune des actions étaient exécutées *séquentiellement* dans leur ordre d'apparition.

Dans la vie réelle, il existe évidemment une infinité d'exemples où le traitement à effectuer est variable, selon la nature des données.

Les langages de programmation impératifs donnent au programmeur, à travers la structure de contrôle alternative, la possibilité de procéder à des traitements spécifiques.

La structure alternative

- Syntaxe

```
If condition Then
   [statements]
{ElseIf condition-n Then
   [elseifstatements] }
[Else
   [elsestatements]]
End If
```

Dans l'instruction conditionnelle, si l'évaluation de la condition donne comme résultat la valeur logique TRUE , alors ce sont les instructions suivant le mot réservé THEN qui sont exécutées et l'exécution continue en séquence avec l'instruction qui suit logiquement l'instruction IF END IF. Les parties ELSEIF, ELSE, même si elles sont présentes, ne seront pas exécutées.

Il peut y avoir autant de ELSEIF que nécessaire, mais il ne peut y en avoir après le ELSE.

Si l'évaluation de l'expression donne la valeur FALSE, alors ce sont toutes les instructions ELSEIF (si présentes et dans l'ordre d'apparition) qui sont évaluées. Si une des ces conditions est évaluée à TRUE, les instructions associées sont exécutées et l'exécution continue en séquence avec l'instruction qui suit logiquement l'instruction IF END IF.

Dans le cas où la condition du IF est évaluéeé à FALSE et que toutes les conditions ELSEIF éventuelles sont aussi évaluées à FALSE, les instructions associées à la partie ELSE, pourvu qu'elle existe, sont exécutées.

Dans le cas où il n'y a pas de partie ELSE, l'exécution continue en séquence avec l'instruction qui suit logiquement l'instruction IF.

Il est permis d'écrire des instructions IF imbriquées (une instruction IF contenant une autre instruction IF, etc.).

Il est important d'écrire proprement les instructions IF en mettant en évidence les imbrications!

La structure alternative

- Exemples :
  - les programmes convertissant les miles en km acceptent des nombres négatifs
  - en cliquant sur le bouton Cancel, le programme se "plante"

Les programmes précedents calculent le résultat exact pourvu que l'utilisateur entre une donnée correcte.

Tel n'est pas le cas s'il entre par exemple un nombre de miles négatif, ou s'il entre des caractères ou s'il clique sur le bouton "Cancel" dans la boîte de dialogue "InputBox".

Les deux programmes doivent être modifiés afin d'éliminer ce comportement incorrect.

La structure alternative

deux instructions IF imbriquées !

```
Private Sub BoutonCalcul_Click()
   Dim miles As Variant, km As Double
   Dim res As Integer
   ' saisie de la donnée
   miles = Val(InputBox("Entrez le nombre de miles", "Conversion de miles en
     kilomètres"))
   If miles <> 0 Then ' l'utilisateur a cliqué OK
      ' vérification de la validité de la donnée
      If miles < 0 Then
         res = MsgBox("Nombre de miles négatif", vbCritical, "Résultat de la conversion")
      Else
         ' calcul du résultat
         km = 1.609 * miles
         ' affichage du résultat
         res = MsgBox(miles & " miles équivalent à " & km & " kilomètres", vbOKOnly,
     "Erreur")
      End If
   End If
End Sub
```

La variable *miles* n'est pas saisie directement comme un nombre, mais d'abord comme du texte qui est transformé par la suite, grâce à la fonction *Val*(), en un nombre.

Exemples :

```
Dim MyValue
MyValue = Val("2457")              ' Returns 2457.
MyValue = Val(" 2 45 7")           ' Returns 2457.
MyValue = Val("24 and 57")         ' Returns 24.
MyValue = Val("a24")               ' Returns 0.
MyValue = Val("2a24 ")             ' Returns 2.
MyValue = Val("")                  ' Returns 0.
```

La fonction *InputBox*() renvoie la valeur entrée par l'utilisateur si celui-ci clique sur "Ok" ou la chaîne "" (vide) s'il clique sur "Cancel". Ce comportement permet au programmeur de savoir quel bouton a été cliqué.

La structure alternative

```
If Val(miles.Value) >= 0 Then
   km.Value = 1.609 * Val(miles.Value)
Else
   Dim res As Integer
   res = MsgBox("Nombre de miles négatif", _
            vbCritical, _
            "Erreur")
End If
```

La structure alternative

- Exercice: calcul de la date du lendemain
  - Il s'agit d'écrire un programme qui calcule la date du lendemain à partir d'une *date correcte* entrée au clavier.
  - Ce problème d'apparence simple n'est pas si anodin que cela, car il montre une des difficultés de la programmation : la précision des concepts manipulés par le programme.

Description des données et résultats :

année entrée	*a*
mois entré	*m*
jour entré	*j*
année du lendemain	*al*
mois du lendemain	*ml*
jour du lendemain	*jl*

La structure alternative

```
si pas fin de mois
   alors { passer simplement au jour suivant }
        jl ← j + 1        ml ← m          al ← a
   sinon { c'est la fin d'un mois. Est-ce la fin d'année  ? }
        si pas fin d'année
                alors { c'est une fin de mois normale }
                        jl ← 1   ml ← m + 1     al ← a
                sinon { c'est bien la St. Sylvestre }
                        jl ← 1   ml ← 1 al ← a + 1
        finsi
finsi
```

Il ne reste qu'à formaliser les différentes conditions. La condition *fin de mois* peut être écrite ainsi :

j = 31

ou *j* = 30 et *m* = 4, 6, 9 ou 11

ou *j* = 29 et *m* = 2

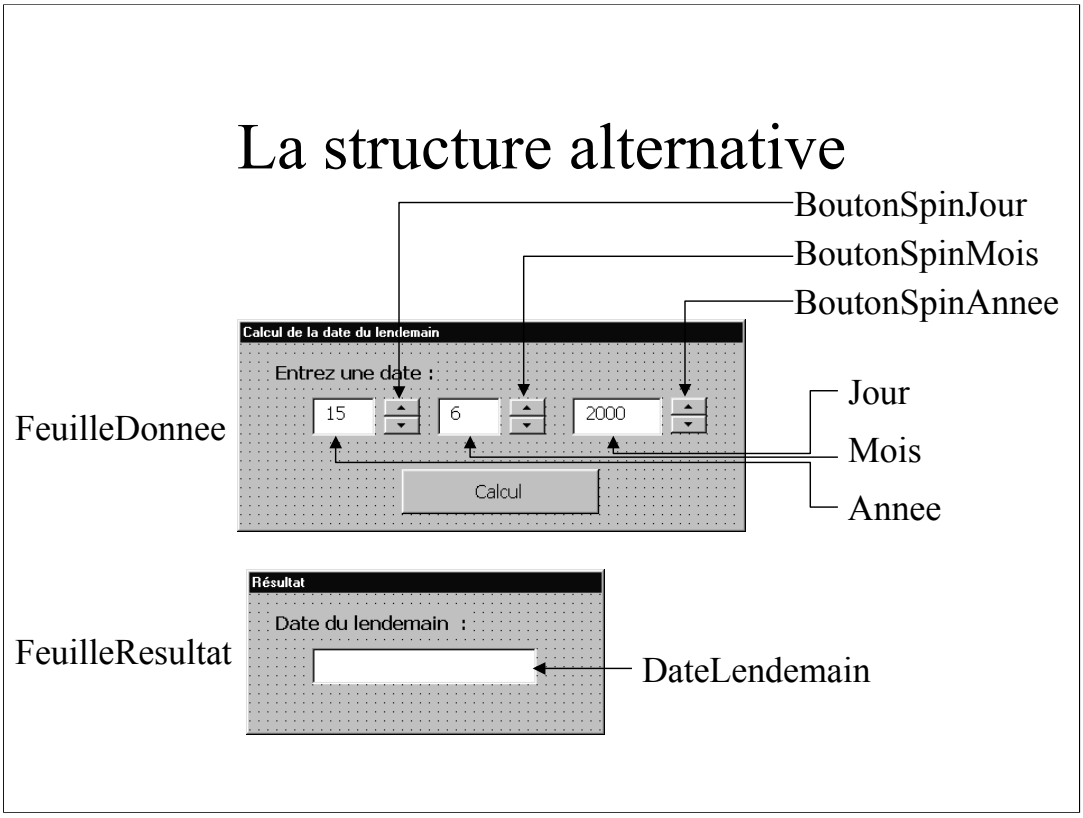
ou *j* = 28 et *m* = 2 et ce n'est pas une année bissextile

Remarquons encore une fois que nous avons supposé que la date entrée est correcte.

La condition *année bissextile* peut, quant à elle, être formalisée de la façon suivante :

((*a* divisible par 4) et (*a* pas divisible par 100)) ou (*a* divisible par 400)

La condition *fin d'année* s'écrit : *fin de mois* et *m* = 12

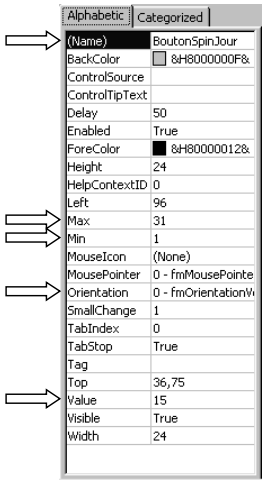


Pour la réalisation de l'interface utilisateur du programme, nous prévoyons deux "userform" appelés respectivement *FeuilleDonnee* et *FeuilleResultat*.

FeuilleDonnee présente trois contrôles "TextBox" associés chacun à un contrôle "SpinButton".

Dans *FeuilleDonnee* l'utilisateur fixe une date, puis clique sur le bouton "Calcul" pour voir afficher *FeuilleResultat.*

La structure alternative



- dans les propriétés de chaque contrôle "SpinButton", nous fixons
  - le nom
  - la valeur logique Max
  - la valeur logique Min
  - l'orientation
  - la valeur de départ

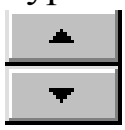
Un contrôle "SpinButton" peut être présenté horizontalement ou verticalement. Les valeurs Max et Min sont des limites logiques autorisées.

Mais attention : le système ne fait aucun contrôle pour vérifier qu'il y a dépassement de ces valeurs limites, c'est au programmeur que revient cette charge !

Cela est tout à fait logique, puisque jusqu'à présent, nous n'avons pas encore associé de zone de saisie, en l'occurrence un contrôle de type "TextBox" au "SpinButton".

La structure alternative

- Pour chacun des contrôles de type "SpinButton", nous devons définir ce qu'il doit faire en présence des événements du type
  - SpinDown
  - SpinUp



La structure alternative

```
Private Sub BoutonSpinJour_SpinUp()
   If Jour.Value < BoutonSpinJour.Max
    Then
      Jour.Value = Jour.Value + 1
   Else
      Jour.Value = BoutonSpinJour.Min
   End If
End Sub
```

```
Private Sub BoutonSpinJour_SpinDown()
   If Jour.Value > BoutonSpinJour.Min Then
      Jour.Value = Jour.Value - 1
   Else
      Jour.Value = BoutonSpinJour.Max
   End If
End Sub
```

Lorsque l'utilisateur clique sur la flèche montante du contrôle *BoutonSpinJour*, c'est clairement pour faire avancer de "un" la valeur du contrôle *Jour*.

Le programmeur doit donc programmer cette action en veillant à ce que cette valeur ne dépasse pas 31. Plutôt que de travailler directement avec la constante numérique 31, nous travaillons avec la valeur conservée dans une propriété de *BoutonSpinJour*, à savoir *Max*. Lorsque que la valeur limite supérieure est atteinte, nous recommençons avec la valeur limite inférieure !

Il en est de même si l'utilisateur clique sur la flèche descendante du contrôle *BoutonSpinJour* ; c'est clairement pour faire reculer de "un" la valeur du contrôle *Jour*.

```
Private Sub BoutonCalcul_Click()
   Dim a As Integer, m As Integer, j As Integer
   Dim al As Integer, ml As Integer, jl As Integer

   Dim annee_divisible_par_4 As Boolean, annee_divisible_par_100 As Boolean
   Dim annee_divisible_par_400 As Boolean

   a = Annee.Value
   m = Mois.Value
   j = Jour.Value

   annee_divisible_par_4 = IIf(a Mod 4 = 0, True, False)
   annee_divisible_par_100 = IIf(a Mod 100 = 0, True, False)
   annee_divisible_par_400 = IIf(a Mod 400 = 0, True, False)

   ' calcul de la date du lendemain
   If Not ((j = 31) Or _
      ((j = 30) And ((m = 4) Or (m = 6) Or (m = 9) Or (m = 11))) Or _
      ((j = 29) And (m = 2)) Or _
      ((j = 28) And (m = 2) And _
            Not ((annee_divisible_par_4 And _
                Not annee_divisible_par_100) Or _
                annee_divisible_par_400) _
      ) _
     ) Then      ' on n'est pas à la fin d'un mois
                jl = j + 1
                ml = m
                al = a
      ElseIf m <> 12 Then      ' fin de mois mais pas fin d'année
                al = a
                ml = m + 1
                jl = 1
      Else ' c'est la St. Sylvestre
                al = a + 1
                ml = 1
                jl = 1
      End If
      FeuilleResultat.DateLendemain.Value = jl & "-" & ml & "-" & al
      FeuilleResultat.Show
End Sub
```

La structure alternative

- Forme spéciale d'instruction alternative, la fonction :
  - IIF(expression, expr_Vrai, expr_Faux)

Si *expression* est évaluée à TRUE, la fonction IIF() renvoie comme valeur le résultat de l'évaluation de l'expression *expr_VRAI*, autrement la valeur *expr_FAUX*.

Exemple :

```
mention = iif(note > 16, "Très bien", "satisfaisant")
```

Attention : *expr_VRAI* et *expr_FAUX* sont évalués toujours tous les deux !

La structure répétitive

- Pour introduire cette structure de contrôle, nous réalisons un programme qui calcule la somme des *n* premiers nombres entiers positifs, où *n* entier positif ou nul est entré au clavier de l'ordinateur.
- En clair, il s'agit de calculer la somme : 1+ 2 + 3 + … + *n*, mais en supposant que nous n'ayons pas la formule pour la calculer directement.

Description des données et des résultats :

entier positif ou nul entré au clavier	*n*
somme des entiers	*somme*

Méthode de résolution :

1. entrer *n*
2. calculer *somme*
3. afficher *n* et *somme*

Raffiner 2 :

Pour obtenir la somme, il suffit d'ajouter successivement le nombre suivant *i* à la somme déjà calculée. Initialement *somme* vaut 0 et *i* vaut 1. Le nombre entier suivant se trouve dans la variable *i*. Cette répétition s'arrête lorsque $i > n$.

somme ← 0

i ← 1

tant que $i \leq n$ faire ┐ initialisations

    somme ← *somme* + *i* ┘

    i ← *i* + 1

fintantque ┘ corps de la boucle

La structure répétitive

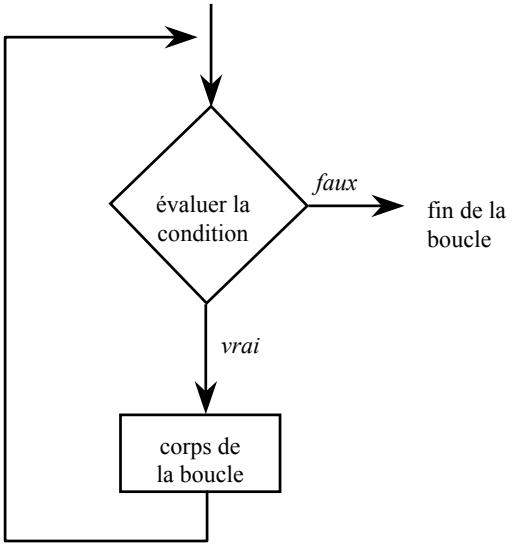
- La structure de contrôle de répétition s'écrit:

tant que *condition de maintien* faire
  corps de la boucle
fintantque

La sémantique de la structure de contrôle « tant que … fintantque » est celle-ci :

> la condition de maintien dans la boucle est d'abord évaluée. Si elle est vérifiée, c'est-à-dire si sa valeur est *vrai*, le corps du « tant que … fintantque » est exécuté. Après exécution de la dernière instruction de ce corps, l'exécution continue avec l'évaluation de la condition de maintien, d'où l'effet de boucle ou répétition.

La boucle ou répétitive s'arrête lorsque la condition d'arrêt de la boucle est vérifiée. La condition qui apparaît dans le « tant que … fintantque » est une condition de maintien dans la boucle. La condition d'arrêt de la boucle s'obtient donc en niant la condition de maintien ou, inversement, la condition de maintien s'obtient en niant la condition d'arrêt.



La structure répétitive

- Attention :

  à partir de maintenant, il est possible d'écrire des programmes pour lesquels l'exécution ne s'arrête jamais (sauf intervention externe) !

Comme le corps d'une boucle est exécuté jusqu'à la réalisation de la condition d'arrêt, il est primordial, lorsque l'on écrit une boucle, de s'assurer que le corps de la boucle contienne des instructions qui garantissent effectivement la possibilité de la réalisation de la condition d'arrêt. Si tel n'était pas le cas, alors le programme contenant la boucle ne s'arrêterait jamais de lui-même vu que la condition d'arrêt ne serait jamais réalisée. Lorsque l'on est dans un cas pareil, on dit que le programme « boucle » ou qu'il est dans une boucle infinie. Voici un exemple simple de boucle infinie :

$i \leftarrow 1$

tant que $i < 10$ faire

    afficher('Coucou')

fintantque

Vu que *i* vaut 1 initialement, le corps de la boucle est exécuté. Ensuite la condition est réévaluée ; comme *i* n'a pas changé, la condition donne de nouveau *vrai* et ainsi de suite. Le corps de cette boucle ne contient pas d'instruction pouvant modifier un des opérandes de la condition d'arrêt. Si on programme cet algorithme et qu'on l'exécute sur machine, il faut généralement une intervention externe de l'utilisateur pour l'arrêter, au pis en redémarrant la machine (sur un système simple).

La structure répétitive

- Format VBA :

```
"WHILE" expression
        instruction(s)
"WEND"
```

Tant que l'expression est évaluée à la valeur *vrai*, la ou les instructions qui suivent l'expression sont exécutées.

Si l'expression est évaluée à *faux*, l'instruction *while* est terminée et l'exécution continue dans le programme avec l'instruction suivant logiquement l'instruction *wend*.

Ainsi, si à la première évaluation de l'expression, celle-ci a comme valeur *faux*, le bloc d'instructions compris entre WHILE et WEND n'est jamais exécuté.

La structure répétitive

```
Private Sub BoutonDemarrage_Click()
   Dim n As Integer, somme As Integer, i As Integer

   n = InputBox("Entrez un entier positif", _
                "Calcul de la somme des N premiers entiers", 0)

   i = 1
   somme = 0
   While i <= n
      somme = somme + i
      i = i + 1
   Wend

   MsgBox ("La somme des " & n & " premiers entiers vaut " & somme)
End Sub
```

Remarquons que nous écrirons toujours le corps d'une boucle légèrement décalé vers la droite afin de bien montrer la portée de la structure de contrôle.

Il reste une remarque à faire sur l'obtention de la condition d'arrêt à partir de la condition de maintien ou vice versa. Comme nous l'avons vu plus haut, une condition s'obtient en niant l'autre. Pour ce faire, on peut bien entendu utiliser l'opérateur logique NOT.

Mais on peut aussi utiliser les lois de « de Morgan » qui s'écrivent :

non(A ou B) = non(A) et non(B)

non(A et B) = non(A) ou non(B)

Supposons, par exemple, que la condition d'arrêt d'une boucle s'écrive :

$(x \leq 3)$ ou $(x \geq 5)$

La condition de maintien dans la boucle est donc :

non$[(x \leq 3)$ ou $(x \geq 5)]$ = non$(x \leq 3)$ et non$(x \geq 5)$ = $(x > 3)$ et $(x < 5)$

La structure répétitive

- Calculer la somme d'une suite de nombres positifs ou nuls entrés au clavier de l'ordinateur.
- La fin de la suite est marquée par un nombre négatif.

Le problème précédent était un peu spécifique en ce sens que la condition d'arrêt de la boucle ne dépendait que d'un paramètre fixé une fois pour toutes avant l'exécution de la boucle, lors de la saisie du nombre entier.

On peut aussi être confronté à la situation où le facteur déterminant la condition d'arrêt de la boucle est lui-même déterminé dans le corps de la boucle.

Nous allons étudier ce cas sur un petit programme qui calcule la somme d'une suite de nombres positifs ou nuls entrés au clavier de l'ordinateur. La fin de la suite est marquée par un nombre négatif.

Pour résoudre ce problème, il faut additionner la dernière valeur lue à la somme des valeurs déjà traitées. L'historique de ces valeurs ne nous intéresse pas et il suffit d'une variable contenant la dernière valeur lue et non encore traitée.

Description des données et des résultats :

dernière valeur lue et non encore traitée	*nbre*
somme des valeurs traitées	*somme*

Méthode de résolution :

Il est clair qu'il faut utiliser une boucle pour résoudre ce problème. Le corps de boucle s'écrit :

somme ← *somme* + *nbre*
lire(*nbre*)

La condition d'arrêt de la boucle est : *nbre* < 0, c'est-à-dire que l'on arrête si l'utilisateur entre un nombre négatif.

Pour que la boucle fonctionne correctement, il faut initialiser la somme à zéro et lire le premier nombre. Nous obtenons ainsi :

```
somme ← 0
lire(nbre)          { saisie d'avance : valeur lue et non encore traitée }
tant que nbre ≥ 0 faire
    somme ← somme  + nbre
    lire(nbre)     { saisie courante : valeur lue et non encore traitée }
fintantque
afficher(somme)
```

Cet algorithme révèle donc qu'il y a deux opérations de saisie, la saisie d'avance et la saisie courante. Cela peut paraître surprenant à première vue, mais en fait c'est normal puisque c'est la valeur lue qui détermine l'arrêt de la boucle.

On aurait pu être tenté de changer l'ordre des instructions du corps de la boucle.

lire(*nbre*)
somme ← *somme* + *nbre*

La condition d'arrêt de la boucle ne change pas, c'est toujours *nbre* < 0. Alors que faire si *nbre* est négatif ? On doit empêcher l'addition de ce *nbre* à *somme*, ce qui peut se faire par un test. On obtient :

lire(*nbre*)
si *nbre* ≥ 0 alors
    somme ← *somme* + *nbre*
finsi

L'initialisation de *somme* ne change pas, par contre pour *nbre*, il y a un problème. Il ne reste qu'à utiliser un « truc » pour que ça marche quand même : on initialise *nbre* à zéro ce qui permet d'entrer dans le corps de la boucle :

Nous obtiendrions l'algorithme suivant :

somme ← 0
nbre ← 0
tant que *nbre* ≥ 0 faire
    lire(*nbre*)
    si *nbre* ≥ 0 alors
        somme ← *somme* + *nbre*
    finsi
fintantque
afficher(*somme*)

Cet algorithme est mauvais car il nous force à utiliser des trucs pour empêcher l'arrêt prématuré de la boucle ; de plus, le corps de la boucle est inutilement compliqué à cause de la structure alternative qui n'est pas économique du tout car le test si *nbre* ≥ 0 se fait deux fois pour chaque passage dans la boucle ! Une solution pareille doit donc être rejetée.

La structure répétitive

```
Dim nbre As Integer, somme As Integer
somme = 0
' saisie d'avance
nbre = InputBox("Entrez un entier positif", _
            "Calcul de la somme de N nombres ( < 0 pour arrêter)", 0)
While nbre >= 0
   somme = somme + nbre
   ' saisie courante
   nbre = InputBox("Entrez un entier positif", _
               "Calcul de la somme de N nombres ( < 0 pour arrêter)", 0)
Wend

MsgBox ("La somme des nombres entrés vaut " & somme)
```

La structure répétitive

- VBA offre de multiples formes de structures de contrôle de répétition
  - WHILE…WEND
  - FOR…NEXT
  - FOR EACH…NEXT
  - DO…LOOP

La plupart des langages de programmation offrent au programmeur plusieurs structures de contrôle répétitive mieux adaptées à l'une ou l'autre situation.

WHILE…WEND est la structure la plus générale que l'on rencontre dans tous les langages de programmation usuels.

Lorsque que le nombre d'itérations est connu, fixé à l'avance, on peut utiliser la boucle FOR…NEXT bien adaptée à cette situation.

La boucle FOR EACH…NEXT est adaptée à des situations que nous verrons plus tard.

Finalement, la boucle DO …LOOP existe en deux formes fondamentales et est très flexible.

La structure répétitive

- FOR variable = debut TO fin [STEP pas]
  {instruction}
  [EXIT FOR]
  {instruction}
- NEXT [variable]

variable	Requis. Variable numérique utilisée comme *compteur* de boucle. Elle ne peut être un booléen ou un élément d'un tableau.
debut	Requis. Valeur initiale du compteur.
fin	Requis. Valeur finale du compteur..
pas	Optionnel. La quantité ajoutée au compteur après chaque passage dans la boucle. Si STEP est absent, le pas vaut 1.

Le pas peut être positif or négatif. S'il est >= 0, le corps de la boucle est exécuté tant que *compteur* <= *fin*. S'il est < 0, le corps de la boucle est exécuté tant que *compteur* >= *fin*.

Lorsque toutes les instructions du corps de boucle ont été exécutées, *fin* est additionné automatiquement à la variable *compteur*.

EXIT FOR est prévu pour quitter prématuréement la boucle et est généralement utilisé conjointement avec une instruction IF !

Attention : Il est fortement déconseillé de changer la valeur d'une variable compteur à l'intérieur de la boucle !

La structure répétitive

Exemple : Calcul de la factorielle d'un nombre

factorielle(0) = 1
factorielle(n) = n*(n-1)*(n-2)*…3*2*1

```
Dim nbre As Integer, fact As Integer, i As Integer

nbre = InputBox("Entrez un entier positif ou nul", _
        "Calcul de la factorielle d'un nombre", 0)

fact = 1
For i = nbre To 2 Step -1
   fact = fact * i
Next i

MsgBox ("La factorielle de " & nbre & " vaut " & fact)
```

La structure répétitive

L'instruction de répétition existe sous quatre formes différentes :

```
Do  While condition
      {instruction}
      [Exit Do]
      {instruction}
Loop
```

```
Do  Until condition
      {instruction}
      [Exit Do]
      {instruction}
Loop
```

La forme DO WHILE condition …LOOP est équivalente à la forme WHILE …WEND. La condition est une *condition de maintien* dans la boucle !

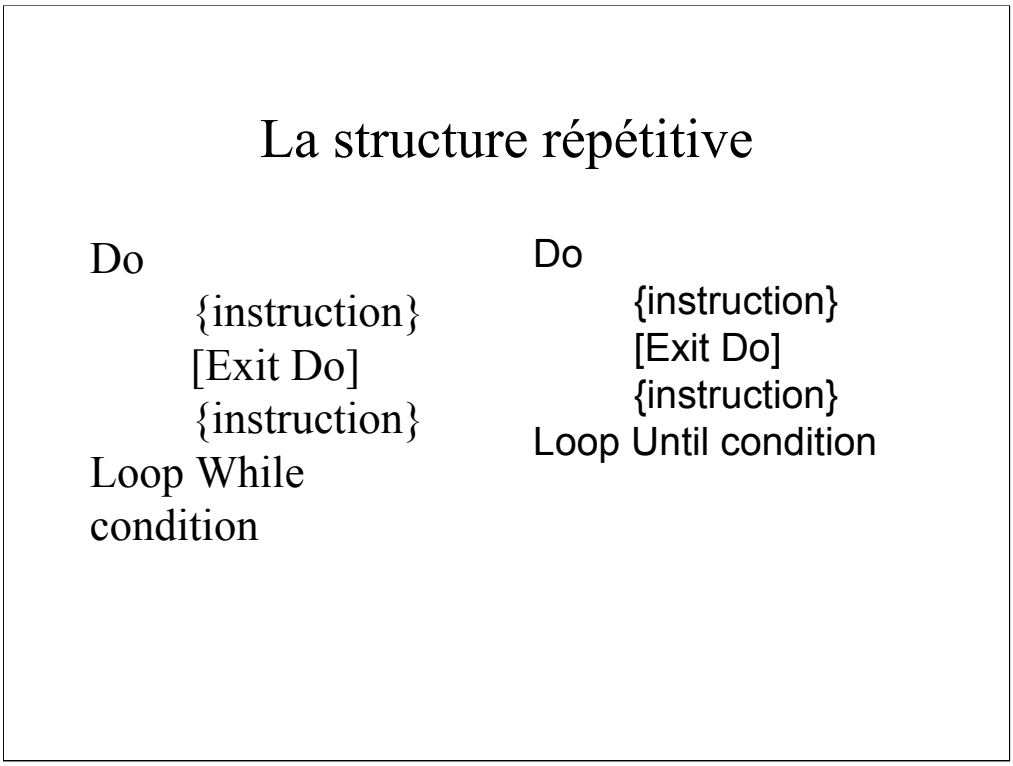
Si nous reprenons le premier exemple de ce chapitre, on écrit :

```
i = 1
somme = 0
Do While i <= n
   somme = somme + i
   i = i + 1
Loop
```

DO UNTIL condition …LOOP est très semblable sauf que cette fois-ci la condition est la *condition d'arrêt* de la boucle. Exemple :

```
i = 1
somme = 0
Do Until i > n
   somme = somme + i
   i = i + 1
Loop
```

Dans les deux cas, le corps de la boucle peut ne jamais être exécuté !



Ces deux formes se distinguent fortement des deux précédentes, car cette fois, le corps de la boucle est toujours exécuté au moins une fois, car la vérification de la condition de maintien ou d'arrêt se fait chaque fois à la fin de l'exécution du corps !

Cette forme peut être très pratique dans certains cas, comme par exemple la vérification d'une saisie.

```
Do
   n = InputBox("Entrez un entier positif", _
                "Calcul de la somme des N premiers entiers", 0)
Loop Until n >= 0
```

ou

```
Do
   n = InputBox("Entrez un entier positif", _
                "Calcul de la somme des N premiers entiers", 0)
Loop While n < 0
```

Les fonctions

- Trois structures de contrôle fondamentales :
  - la structure séquentielle
  - la structure alternative
  - la structure répétitive
- Méthode de développement par raffinements successifs : nécessité d'un outil de structuration du code

Jusqu'à présent, nous avons étudié les trois structures de contrôle fondamentales permettant d'écrire tout programme calculable effectivement. Rappelons encore une fois ces trois structures :

• la structure séquentielle: les instructions sont exécutées en séquence, l'une après l'autre, dans leur ordre d'apparition ;

• la structure alternative: les instructions sont exécutées en fonction de la réalisation d'une condition;

• la structure répétitive: les instructions sont exécutées un nombre variable de fois dépendant de la réalisation de la condition d'arrêt.

Mais l'écriture de programmes *corrects* n'est pas une chose aussi facile que cela paraît à première vue. Il est intéressant de disposer d'une méthode de développement permettant de construire des programmes corrects. C'est ce que nous avons fait dans les chapitres précédents en utilisant la méthode de développement de programmes par raffinements successifs tout en s'aidant d'une méthode de construction de boucles basée sur le concept d'invariant.

Dans la méthode de développement par raffinements successifs, le but est de décomposer un problème en sous-problèmes, chacun de ces sous-problèmes étant plus facile à résoudre que le problème initial. C'est ce que nous avons fait dans les programmes réalisés, mais l'inconvénient est que le programme ne garde plus aucune trace de cet effort de structuration et de décomposition, à part les commentaires.

Les fonctions

- Outil de structuration du code :
  - le sous-programme
    - fonction
    - procédure
- Ce sont des concepts que l'on retrouve dans tous les langages de programmation impératifs.

Cette lacune est comblée par le concept de sous-programme (*procedure*, *subroutine*). Nous verrons qu'il existe deux types de sous-programmes : les fonctions et les procédures.

Ces notions favorisent, supportent le style de développement par raffinements successifs en ce sens qu'elles permettent au programmeur de traduire en VBA, telles quelles, les différentes phases de raffinement.

Le concept de fonction ou procédure a encore d'autres avantages. Il permet notamment de pouvoir réutiliser du code existant et, de ce fait, augmente la productivité. De plus, il permet d'introduire un niveau d'abstraction en permettant de construire et de manipuler des structures qui n'existent pas d'office dans le langage.

Le concept de fonction et/ou de procédure est présent dans tous les langages de programmation impératifs usuels.

Les fonctions

- sous-programme :
  - ensemble d'instructions, calculant un certain nombre de résultats en fonction d'un certain nombre de données.
  - les données et résultats du sous-programme sont appelés arguments ou paramètres.

Un sous-programme est un moyen dont dispose le programmeur pour nommer une action complexe.

Cette action est décrite au moyen d'un ensemble d'instructions du langage. Le programmeur a le moyen de paramétriser cette action à travers des arguments qu'il fournit au sous-programme.

Pour réaliser l'action décrite par un sous-programme, un programme ou un sous-programme peut *appeler* ou *invoquer* le sous-programme en question.

La différence essentielle entre un programme et un sous-programme est donc qu'un sous-programme peut agir pour le compte d'un programme ou d'un sous-programme.

Lorsqu'il n'y a pas de confusion possible, nous utiliserons indifféremment les termes programme et sous-programme.

Lorsqu'un programme appelle un sous-programme, nous distinguerons entre le *programme appelé* et le *programme appelant*. A la fin de l'exécution du programme appelé se produit un retour automatique vers le programme appelant à l'instruction suivant logiquement l'instruction d'appel du sous-programme.

Les fonctions

- la définition d'un sous-programme comporte :
  - le nom du sous-programme
  - le nombre et le type de ses arguments
  - une description de l'action

Lorsque le programmeur définit un sous-programme, il doit l'identifier en lui donnant un nom et en précisant le nombre et le type de ses arguments. Ensuite, il fournit la suite d'instructions produisant l'action voulue au moment de l'exécution du sous-programme.

Ces instructions manipulent généralement les arguments éventuels du sous-programme. Il est donc nécessaire de donner un nom à chacun des arguments du sous-programme. On les appelle *arguments formels* ou *paramètres formels*.

Les fonctions

- appel d'un sous-programme :
  - écrire le nom du sous-programme suivi d'une liste d'arguments effectifs

Pour appeler un sous-programme, il suffit, dans le programme appelant, d'écrire le nom du sous-programme suivi d'une liste éventuelle d'arguments effectifs (réels).

Chacun de ces arguments effectifs est censé remplacer l'argument formel correspondant tel qu'il est défini dans la définition du sous-programme. Nous verrons plus loin comment se fait très précisément ce remplacement. Dans la plupart des langages de programmation impératifs, la correspondance entre les arguments effectifs et formels doit être totale, tant du point de vue nombre que de leurs types respectifs.

Au moment de l'exécution, dans le programme appelant, à l'endroit de l'appel du sous-programme, les instructions de ce dernier seront exécutées avec les arguments effectifs fournis dans l'appel.

Une fonction qui s'appelle elle-même est appelée une *fonction récursive*.

Les fonctions

- un sous-programme ne représentant qu'une action est appelé une procédure
- un sous-programme dénotant une valeur est une fonction

A travers l'utilisation de procédures, le programmeur a la possibilité de définir de nouvelles primitives (instructions). A la différence des instructions prédéfinies dans le langage, ces nouvelles instructions ne sont comprises que par le programme dans lequel ces procédures sont définies.

Une fonction ne peut être appelée que dans une expression. Il faut évidemment que la fonction en question soit définie dans le programme appelant !

Les fonctions

- dans VBA

```
Function nom_fonction [(arglist)] [As type]
   {   instruction | nom_fonction = expression |
       Exit Function }
End Function
```

Une fonction a un nom et possède éventuellement une liste d'arguments placée entre parenthèses. Le programmeur peut indiquer le type de la valeur retournée par la fonction à travers le mot-clé AS.

Le corps de la fonction délimitée par FUNCTION et END FUNCTION se compose d'instructions VBA qui peuvent être bien sûr des appels de fonction/procédure. L'instruction EXIT FUNCTION permet d'arrêter l'exécution de la fonction et de continuer l'exécution à l'endroit qui suit immédiatement l'appel de la fonction. Normalement cela est réalisé automatiquement lorsque l'exécution arrive à END FUNCTION.

La valeur retournée par la fonction est donnée par une instruction d'affectation où la zone réceptrice porte le même nom que la fonction.

Une fonction peut contenir plusieurs instructions EXIT FUNCTION, mais du point de vue méthodologique, cette pratique est à déconseiller.

Le principe sain qui prévaut est : une fonction a un point d’entrée et un point de sortie !

Les fonctions

- dans VBA

```
Sub nom_fonction [(arglist)]
   { instruction}
   [ Exit Sub ]
   { instruction}
End Sub
```

Une procédure a un nom et possède éventuellement une liste d'arguments placée entre parenthèses.

Le corps de la fonction délimitée par SUB et END SUB se compose d'instructions VBA qui peuvent être bien sûr des appels de fonction/procédure. L'instruction EXIT SUB permet d'arrêter l'exécution de la procédure et de continuer l'exécution à l'instruction qui suit logiquement l'appel de la procédure. Normalement cela est réalisé automatiquement lorsque l'exécution arrive à END SUB.

Une fonction peut contenir plusieurs instructions EXIT SUB, mais du point de vue méthodologique, cette pratique est à déconseiller.

Les fonctions

- La syntaxe simplifiée d'un argument de la liste d'arguments est la suivante :

  [ByVal | ByRef] varname As type

- deux arguments sont séparés par une virgule

Un argument est passé par défaut par référence, c'est-à-dire que l'argument dans la fonction est un alias du paramètre effectif. En clair, si la fonction modifie l'argument, c'est en réalité l'argument effectif qui est changé. C'est le mode de passage par défaut.

Lorsque le programmeur passe un argument par valeur, alors la fonction dispose de sa propre copie de l'argument. Si elle le modifie, cela n'affecte en rien l'argument effectif correspondant.

Les fonctions

- appel d'une procédure
  - comme instruction
    - nom_de_procédure argument1, argument2…
  - ou
    - CALL nom_de_procédure(arg1, arg2…)
- appel d'une fonction
  - dans une expression
    - nom_de_fonction(argument1, argument2…)

Lors de l'appel d'une fonction ou procédure, les arguments effectifs utilisés doivent être compatibles avec les arguments formels tels qu'ils sont définis dans la déclaration de cette fonction ou procédure.

Si le mode de transmission de l'argument est ByRef, l'argument formel qui doit être une variable, est lié à l'argument effectif, il en devient un alias.

Si le mode de transmission de l'argument est ByVal, l'argument effectif, qui est alors une expression, est évalué et l'argument formel correspondant aura comme valeur initiale la valeur résultant de cette évaluation.

En principe, le nombre et le type des paramètres effectifs et formels doivent correspondre.

Les fonctions

- dans une feuille de travail normale Excel
  - une fonction définie par l'utilisateur peut être utilisée tout comme une fonction prédéfinie
  - une procédure définie par l'utilisateur peut être associée à une macro

Un avantage des fonctions et des procédures définies par l'utilisateur avec VBA est qu'elles peuvent être utilisées dans des feuilles de calcul Excel.

Les fonctions

- une fonction /procédure peut contenir des déclarations de variables
- ce sont des variables locales à la fonction/procédure
- elles sont automatiquement créées au moment de l'appel et détruites à la fin de l'exécution de la fonction /procédure

Une variable locale peut porter le même nom qu'un variable définie globalement, c'est-à-dire en dehors d'une fonction ou procédure. Une telle variable est appelée une *variable globale*.

Dans le cas où une variable locale porte le même nom qu'une variable globale, cette dernière n'est plus accessible dans la fonction qui définit cette variable locale.

Les fonctions

```
Function Reverse(ByVal s As String) As String
   Dim rev As String, i As Integer

   rev = ""
   For i = Len(s) To 1 Step -1
      rev = rev & Mid(s, i, 1)
   Next i
   Reverse = rev
End Function
```

La fonction Reverse renvoie comme valeur la chaîne de caractères transmise renversée. L'argument effectif n'est pas modifié !

Les fonctions

```
Function DeuxiemeDegre(ByVal a As Double, ByVal b As Double, ByVal c As
    Double, _
                           x1 As Double, x2 As Double) As Boolean
  ' calcul du discriminant
  Dim delta As Double
  delta = b * b - 4 * a * c
  ' détermination des solutions éventuelles
  If (delta < 0) Then
     DeuxiemeDegre = False
  ElseIf (delta = 0) Then
     x1 = -b / (2 * a)
     x2 = x1
     DeuxiemeDegre = True
     ...
     ...
  Else
     x1 = (-b - Sqr(delta)) / (2 * a)
     x2 = (-b + Sqr(delta)) / (2 * a)
     DeuxiemeDegre = True
  End If
End Function
```

Cette fonction admet 5 arguments :

- les 3 premiers sont passés par valeur ; ce sont les données que nous fournissons à la fonction
- le deux derniers sont passés par référence ; ce sont les résultats que la fonction renvoie

De plus, la fonction a comme valeur un booléen indiquant si oui ou non l'équation admet des solutions réelles.

Voici une procédure utilisant cette fonction :

```
Sub test_deuxiemeDegre()
   Dim s1 As Double, s2 As Double
   If DeuxiemeDegre(3, 6, 3, s1, s2) Then
      Debug.Print s1; s2
   End If
End Sub
```

Les tableaux

- un tableau permet de définir un ensemble fini d'éléments de même type
- un tableau est conservé dans la mémoire centrale de l'ordinateur
- tous les éléments d'un tableau portent le même nom, mais chacun porte un numéro (*indice*) différent

Tout programme peut être écrit avec les trois structures de contrôle de base. Néanmoins, nous venons de voir que les concepts de fonction et de procédure peuvent faciliter la mise au point de programmes.

Jusqu'à présent, nous avons manipulé des variables simples. Or dans certains cas, cette approche se révèle assez fastidueuse. C'est pour cela que le concept de tableau est présent dans la plupart des langages de programmation.

Un tableau est une structure de données conservée dans la mémoire centrale de l'ordinateur. L'utilisation d'un tableau permet de définir un ensemble fini d'éléments de même type.

Chaque élément d'un tableau est identifié par le même identificateur (le nom du tableau) et par un numéro unique appelé indice de l'"élément du tableau. Grâce à ces deux données, on peut manipuler chacun des éléments d'un tableau.

Le temps d'accès à un élément est indépendant de la valeur de l'indice, et les éléments peuvent être accédés dans n"importe quel ordre. On dit que c'est une structure de données à accès aléatoire.

Les tableaux

- VBA supporte deux types de tableaux
  - les tableaux statiques
  - les tableaux dynamiques

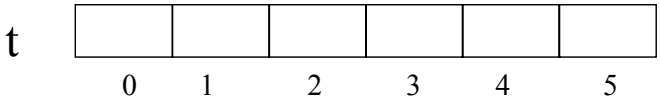
La taille occupée par un tableau statique en mémoire centrale est fixée une fois pour toutes au moment de la compilation.

Pour un tableau dynamique, le programmeur ne doit doit pas spécifier immédiatement la taille que celui-ci occupe en mémoire; ceci se fait plus tard à l'aide de l'instruction REDIM. Attention : cette flexibilité se paie en perte de vitesse d'exécution du programme.

Les tableaux

- Exemple d'un tableau statique :

```
DIM t(5) AS single
```



Tous les éléments d'un même tableau sont du même type !

La taille des tableaux statiques doit être fixée à la compilation. L'indice du premier élément d'un tableau est zéro par défaut. Dans l'exemple ci-dessus, le tableau *t* se compose de 6 éléments numérotés de 0 à 5.

La directive OPTION BASE 1 permet de fixer l'indice du 1er élément des tableaux à 1. Elle doit précéder toute définition de tableau et ne peut figurer qu'une seule fois dans la partie "déclarations" d'un module.

Les éléments d'un tableau sont initialisés à 0 pour les types numériques et à"" (la chaîne vide) pour les chaînes de caractères (string).

Un tableau peut être utilisé comme argument d'un fonction, mais une fonction ne peut pas retourner un tableau autrement que comme un de ses arguments passés par variable.

Les tableaux

- le programmeur doit fixer la borne supérieure du tableau statique
- si la borne inférieure n'est pas indiquée, elle vaut 0 par défaut ou 1 si OPTION BASE 1
- mais il peut aussi fixer celle-ci lui-même
  – exemple : DIM t(-10 TO 10) AS integer

En fixant la borne inférieure d'un tableau statique, le programmeur doit veiller à ce que sa valeur soit inférieure ou égale à celle de la borne supérieure.

```
Erreur :    DIM t(1 TO -1) AS string
Correct :   DIM x(-1 TO 1) AS string    ' 3 éléments d'indice -1, 0, 1
            DIM u(2 TO 2) AS integer    ' 1 élément d'indice 2
```

Les tableaux

- accès à un élément d'un tableau
  Le nom du tableau doit être suivi par une expression placée entre parenthèses.
  L'expression est évaluée à l'exécution et représente l'indice.
- exemple : t(3) = 12.5

```
i = 2
debug.print t(i+1)
```

Une variable indicée représentant un accès à un élément d'un tableau peut s'utiliser partout où une variable normale peut être utilisée.

A l'exécution, il est interdit que l'indice ait une valeur

- plus petite que la borne inférieure du tableau ou
- plus grande que l'indice maximum indiqué dans la définition du tableau.

Si tel est quand même le cas, l'exécution s'arrête avec un message d'erreur.

Remarque : DEBUG.PRINT affiche la valeur de ses arguments dans la fenêtre "Immediate" de l'environnement de développement !

Les tableaux

Exemple :

écrivons un programme qui crée un tableau de nombres, les trie en ordre croissant, puis les affiche dans la fenêtre "Immediate"

```
Option Explicit
Option Base 1
Sub main()
   Const maxValue As Integer = 40
   Const maxArraySize As Integer = 20

   Dim t(maxArraySize) As Single
   Dim i As Integer, j As Integer, ind_max As Integer
   Dim max As Single, tmp As Single

   ' créer les valeurs entières au hasard entre 1 et maxvalue
   For i = LBound(t) To UBound(t)
      t(i) = Int(maxValue * Rnd + 1)
   Next i
```

VBA prévoit deux primitives pour accéder respectivement aux valeurs de la borne supérieure et de la borne inférieure d'un tableau, à savoir UBOUND() et LBOUND(). Elles admettent toutes les deux le tableau comme argument.

VBA, comme la plupart des langages de programmation modernes, supporte le concept de constante. Une constante définit un nom symbolique qui sert à identifier une valeur. Le nom d'une constante peut être utilisé par la suite, dans le domaine de définition de l'identificateur de constante, en lieu et place de la valeur. La valeur attachée au nom symbolique ne peut plus être modifiée dans le programme. Les avantages du concept de constante sont multiples :

• le programme est plus lisible. Au lieu de contenir des valeurs obscures, il contient un texte en clair décrivant au mieux la valeur ;

• le programme est plus facilement maintenable car il suffit en général de modifier la valeur de la constante en un endroit et puis de recompiler le programme. Dans le cas contraire, le programmeur doit vérifier ligne par ligne son code source et corriger les valeurs manuellement, ce qui n'est pas sans risque d'erreur ;

• le fait qu'une valeur soit définie comme une constante interdit au programmeur de changer accidentellement cette valeur. En effet, bien qu'une constante ait un identificateur tout comme une variable, le compilateur ne permet pas de l'utiliser comme une variable et affiche un message d'erreur le cas échéant.

Les tableaux

```
'trier les valeurs
 i = UBound(t)
 While i > LBound(t)
    max = t(1): ind_max = 1
    For j = 2 To i
       If t(j) > max Then
          ind_max = j: max = t(j)
       End If
    Next j
    ' ind_max pointe sur le maximum de 1 à i. Mettons-le à sa
  place
    tmp = t(i): t(i) = t(ind_max): t(ind_max) = tmp
    i = i - 1
 Wend
...
```

```
...
 ' afficher les valeurs
 For i = LBound(t) To UBound(t)
    Debug.Print t(i);
 Next i
 Debug.Print
End Sub
```

La fonction prédéfinie RND() renvoie un nombre pseudo-aléatoire compris entre 0 inclus et 1 exclus.

Il existe beaucoup de façons de trier des éléments. Une méthode simple, mais pas très efficace est le tri par sélection.

Il fonctionne d'après le principe suivant :

- sélectionner l'élément du tableau ayant la plus grande clé ;
- échanger cet élément avec le dernier élément du sous-tableau non encore trié. Un élément vient d'être trié. Appliquer la méthode sur le sous-tableau restant non encore trié.

Remarque : En VBA, il est permis d'écrire plusieurs instructions sur une même ligne si elles sont séparées par le symbole ":".

Les tableaux

- un tableau peut être utilisé comme argument d'une fonction
  - il doit être passé par référence
  - l'argument formel correspondant doit être marqué par une paire ()
- mais : une fonction ne peut avoir comme valeur un tableau

Exemples :

```
Sub init_array(t() As Single)
   Dim i As Integer
   For i = LBound(t) To UBound(t)
      t(i) = Int(maxValue * Rnd + 1)
   Next i
End Sub

Sub display_array(t() As Single)
   Dim i As Integer
   For i = LBound(t) To UBound(t)
      Debug.Print t(i);
   Next i
   Debug.Print
End Sub
```

Les tableaux

Exemple :

```
Sub tri_fonction()
   Const maxArraySize As Integer = 20
   Dim t(maxArraySize) As Single

   ' créer les valeurs entières au hasard
   Call init_array(t)

   'trier les valeurs
   Call sort_array(t)

   ' afficher les valeurs
   display_array t
End Sub
```

```
Sub sort_array(t() As Single)
   Dim i As Integer, j As Integer, ind_max As Integer
   Dim max As Single, tmp As Single
   i = UBound(t)
   While i > LBound(t)
      max = t(1): ind_max = 1
      For j = 2 To i
         If t(j) > max Then
            ind_max = j: max = t(j)
         End If
      Next j
      ' ind_max pointe sur le maximum de 1 à i. Mettons-le à sa place
      tmp = t(i): t(i) = t(ind_max): t(ind_max) = tmp
      i = i - 1
   Wend

End Sub
```

Les tableaux

- un tableau dynamique
  - est un tableau dont la taille n'est pas fixée à l'avance
  - le programmeur peut l'ajuster à sa guise pendant l'exécution
  - MAIS : la modification de la taille d'un tableau est une opération coûteuse !

Il existe des situations où la taille d'un tableau ne peut être estimée convenablement à l'avance.

VBA propose au programmeur le concept de tableau dynamique.

Les tableaux

- déclaration d'un tableau dynamique
  `DIM t() AS single`
  – paire de parenthèses vides !
- ajustage de la taille
  – en préservant les éléments déjà contenus dans le tableau
    • REDIM PRESERVE
  – en effaçant les valeurs déjà présentes
    • REDIM

Pour déclarer un tableau dynamique, il suffit de ne pas indiquer de taille pour le tableau.

Par la suite, si on veut placer des éléments dans le tableau, celui-ci doit être redimensionné à l'aide de l'instruction REDIM afin de réserver de l'espace mémoire pour ces éléments.

Dans l'instruction REDIM, il faut indiquer le nom du tableau ainsi que le nombre d'éléments que l'on souhaite qu'il contienne. En fait, cette instruction provoque, de manière interne, la création d'un nouveau tableau de la taille désirée.

En utilisant l'option PRESERVE, les valeurs existantes sont recopiées de l'ancien tableau dans le nouveau, à hauteur du nombre d'éléments définis dans le nouveau.

Les tableaux

```
Dim t() As Integer, i As Integer
Dim maxValue As Integer

maxValue = InputBox("Combien d'éléments
  ?")
ReDim t(maxValue)

' créer les valeurs entières au hasard
For i = LBound(t) To UBound(t)
   t(i) = Int(maxValue * Rnd + 1)
Next i

' afficher les valeurs
For i = LBound(t) To UBound(t)
   Debug.Print t(i);
Next i
Debug.Print
...
' diminuer la taille du tableau
ReDim Preserve t(4)

' afficher les valeurs
For i = LBound(t) To UBound(t)
   Debug.Print t(i);
Next i
Debug.Print

' augmenter la taille du tableau
ReDim Preserve t(7)

' afficher les valeurs
For i = LBound(t) To UBound(t)
   Debug.Print t(i);
Next i
Debug.Print
```

A l'exécution, ce programme affiche

```
1  4  10  4  3  6  4  5  1  6  3  11
1  4  10  4
1  4  10  4  0  0  0
```

dans la fenêtre "Immediate".

Nous constatons que lorsque le tableau est redimensionné à une valeur plus grande, ces nouveaux éléments sont initialisés à la valeur par défaut de leur type.

Les tableaux

- un tableau peut être multi-dimensionnel
  - jusqu'à 60 dimensions
  - en pratique 2 ou 3 dimensions (matrices, etc.)
- il suffit d'indiquer le nombre d'éléments dans chaque dimension
- dans LBound() et UBound(), il faut indiquer un deuxième argument, le numéro de dimension

Pour définir un tableau multi-dimensionnel, il suffit d'indiquer le nombre d'éléments pour chaque dimension, dans les formats vus pour les tableaux statiques.

Si on veut utiliser les fonctions LBound() et UBound(), on doit utiliser comme deuxième paramètre le numéro de la dimension.

Un tableau dynamique peut aussi être multi-dimensionnel. Mais si on utilise l'instruction REDIM avec l'option PRESERVE, on ne peut modifier dynamiquement le nombre de dimensions et uniquement la taille de la dernière dimension peut être changée.

Par contre, sans PRESERVE, le programmeur peut changer aussi bien le nombre de dimensions que la taille de celles-ci.

Les tableaux

```
Const NbreLignes As Integer = 5
Const NbreColonnes As Integer = 7
Const maxValue As Integer = 40

Dim t(NbreLignes, NbreColonnes) As
  Integer
Dim i As Integer, j As Integer

' créer les valeurs entières au hasard
For i = LBound(t, 1) To UBound(t, 1)
   For j = LBound(t, 2) To UBound(t, 2)
      t(i, j) = Int(maxValue * Rnd + 1)
   Next j
Next i
   ...
   ...
' afficher les valeurs
For i = LBound(t, 1) To UBound(t, 1)
   For j = LBound(t, 2) To UBound(t, 2)
      Debug.Print t(i, j);
   Next j
   Debug.Print
Next i
Debug.Print
```

Ce programme crée une matrice de 5 lignes et 7 colonnes. Elle est initialisée avec des nombres aléatoires et puis la matrice est affichée. Voici une trace d'exécution :

```
 38  40  24  22  25  4  8
 12  1  7  17  22  23  32
 2  17  3  13  25  21  24
 14  37  31  24  18  30  7
 3  10  31  39  14  28  32
```

L'enregistrement

- en VBA, le programmeur peut créer ses propres types en se basant sur les types existants
- un enregistrement (record) se compose d'un nombre fixe d'éléments, de types éventuellement différents.
- ces différents composants et leurs types sont définis par la liste des champs de l'enregistrement.
- l'ordre des champs n'a pas d'importance.

Jusqu'à présent nous avions la possibilité de définir des variables simples de type integer, single,etc. ou alors des variables de type tableau, où chacun des éléments du tableau était du même type.

Avec le concept d'enregistrement, nous avons la possibilité de définir des variables de type complexe par composition (agglomération) de types existants.

Chaque élément faisant partie de l'enregistrement est appelé un champ. Le type du champ est quelconque, pourvu qu'il soit défini. L'ordre dans lequel les champs sont déclarés dans la définition de l'enregistrement n'a pas d'importance.

L'enregistrement

- exemple : les nombres rationnels sont des nombres de la forme a/b où a et b sont deux entiers avec $b \neq 0$.
- comment représenter élégamment ce concept dans VBA ?
- solution : l'enregistrement

Dans VBA, la définition d'enregistrements se fait au niveau global dans un module :

définition_type ::= "TYPE" identificateur définition_record "END TYPE"
définition_record ::= définition_champ { définition_champ }
définition_champ ::= identificateur "AS" identificateur_de_type

Tout comme dans le cas des tableaux, une définition d'enregistrement ne peut être récursive. Ceci signifie que le type d'enregistrement en cours de définition ne peut être utilisé comme type d'un champ de l'enregistrement.

Pour notre exemple, nous écrirons par exemple :

```
Type rationnel
    numerateur As Integer
    denominateur As Integer
End Type
```

L'enregistrement

- l’accès à un champ d’un enregistrement s’effectue par le symbole “.”
- un objet de type enregistrement peut être utilisé comme argument formel/effectif mais doit être passé par référence
- la valeur d’une fonction ne peut être un enregistrement
- il est permis de faire des affectations entre deux objets du même type enregistrement

Pour accéder à un champ d’un enregistrement, il suffit d’indiquer le nom de l’enregistrement, suivi du symbole “.” et du nom du champ en question.

A l’intérieur d’un enregistrement, tous les identificateurs de champs doivent être uniques et ne doivent pas être des mots réservés.

Le symbole “.” peut évidemment être utilisé en cascade si un champ d’un enregistrement est lui-même un enregistrement.

Tout comme pour les tableaux, il est permis d’utiliser comme argument formel/effectif un objet de type *record*, mais une fonction ne peut retourner comme valeur un *record* et un tel objet doit obligatoirement être passé par référence.

Il est de même permis d’effectuer des affectations entre des objets de même type *record*.

L'enregistrement

```
Option Explicit

Type rationnel
   numerateur As Integer
   denominateur As Integer
End Type

Sub init_rat(num As Integer, denom As Integer, r As rationnel)
   r.numerateur = num
   r.denominateur = denom
End Sub

Sub affiche_rat(r As rationnel)
   Debug.Print r.numerateur; "/"; r.denominateur
End Sub
```

L'enregistrement

```
Sub add_rat(r1 As rationnel, r2 As rationnel, r As rationnel)
   r.numerateur = (r1.numerateur * r2.denominateur) + (r2.numerateur * r1.denominateur)
   r.denominateur = r1.denominateur * r2.denominateur
End Sub

Sub main()
   Dim r1 As rationnel, r2 As rationnel, r3 As rationnel
   Call init_rat(1, 3, r1)                    ' affiche 1 / 3
   Call init_rat(2, 5, r2)                    ' affiche 2 / 5
   Call affiche_rat(r1)
   affiche_rat r2

   Call add_rat(r1, r2, r3)

   affiche_rat r3                ' affiche 11 / 15

End Sub
```

Le traitement d'erreurs

- une erreur à l'exécution (fichier non trouvé, division par 0, etc.) provoque l'arrêt brutal du programme
- VBA offre des primitives de gestion d'erreurs d'exécution :
  - On Error GoTo *nom_paragraphe*
  - On Error Resume Next
  - On Error GoTo 0

On Error GoTo *nom_paragraphe*

indique au système qu'en cas d'erreur d'exécution, l'exécution ne doit pas s'arrêter mais continuer à l'endroit indiqué par *nom_paragraphe*. Un paragraphe est une endroit dans une procédure ou fonction dont le début est indiqué par *nom_ paragraphe* suivi de ":".

On Error Resume Next

dans ce cas, on ignore l'erreur et l'exécution continue à l'instruction suivante !

On Error GoTo 0

sert à désactiver le gestionnaire d'erreur instauré par le programmeur et à réactiver le comportement par défaut de VBA.

L'objet prédéfini *Err* permet au programmeur de s'informer sur le type de l'erreur à travers les propriétés *Number* et *Description*.

Le traitement d'erreurs

- Exemple :

```
Public Sub DivisionPar0Geree()
   On Error GoTo GestionnaireErreur

   Dim nbre As Integer
   nbre = InputBox("Entrez un nombre", "Calcul de l'inverse d'un nombre")
   MsgBox ("L'inverse de " & nbre & " = " & 1 / nbre)
   Exit Sub
GestionnaireErreur:
   MsgBox "Tentative de division par 0" & vbNewLine & "Code d'erreur : " &
     Err.Number _
         & vbNewLine & "Message : " & Err.Description
End Sub
```

Dans un gestionnaire d'erreur, le programmeur peut utiliser les instructions suivantes :

Resume

l'instruction ayant provoqué l'erreur est exécutée à nouveau

Resume Next

l'exécution continue à l'instruction suivant logiquement celle qui a provoqué l'erreur

Resume *nom_paragraphe*

l'exécution continue à la première instruction du paragraphe désigné par *nom_paragraphe*

Attention : cette instruction ne peut être utilisée que dans un gestionnaire d'erreurs.

L’utilisation de DLLs

- Une DLL (Dynamic Link Library) est une bibliothèque de fonctions.
- Il existe des DLL système mais le programmeur peut écrire ses propres DLL en C, C++, VB, etc.
- Les DLL système constituent l'API (Application Programming Interface) Windows, donnant l'accès à quasi toutes les fonctions système de Windows.

L'utilisation de fonctions système définies dans des DLL nécessite de bonnes connaissances et une bonne documentation non fournie avec Office de manière standard.

Il peut être intéressant de savoir qu'un programmeur VBA peut aller au-delà des possibilités offertes intrinsèquement par le langage en faisant

- appel à des fonctions système de Windows
- exécuter un travail critique en temps par une routine compilée.

L'utilisation de DLLs

- Une fonction définie dans une DLL doit être déclarée préalablement à son utilisation au niveau global dans un module.

- DECLARE SUB NomSub LIB "NomLib" (ListeParamètres)

- DECLARE FUNCTION NomFunc LIB "NomLib" (ListeParamètres)
  AS NomType

La liste des paramètres peut être précédée par la clause

ALIAS NomAlias

qui désigne le nom effectif de la fonction dans la DLL. Ceci peut être intéressant si ce nom original de la routine est un mot réservé en VBA, débute par un "_", etc.

Il faut faire très attention aux problèmes de correspondance dans la liste des paramètres et du type de la fonction entre les types VBA et les types C/C++ (la plupart des DLL sont réalisées dans ces langages). De ce fait, il faut quasi toujours passer les arguments par valeur (BYVAL), mode de transmission par défaut dans ces deux langages mais pas en VBA !

Ainsi les types C suivants ont les équivalents suivants en VBA

C	**VBA**
BYTE x	ByVal x As Byte
LPBYTE x	x As Byte
short x	ByVal x As Integer
short far * x	x As Integer
LONG x	ByVal x As Long
LPLONG x	x As Long
float x	ByVal x As Long
float far * x	x As Long
double x	ByVal x As Double
double far * x	x As Double

L’utilisation de DLLs

```
Declare Sub RenvoieRepertoireWindows Lib "kernel32" Alias
    "GetWindowsDirectoryA" _
     (ByVal lpBuffer As String, ByVal nSize As Long)

Function NullString(x As String)
   NullString = Left(x, InStr(x, Chr(0)) - 1)
End Function

Function WinDir() As String
   Dim tmp As String
   tmp = Space(256)
   Call RenvoieRepertoireWindows(tmp, 255)
   WinDir = NullString(tmp)
End Function
```

```
Sub AfficheRepertoireWindows()
   MsgBox WinDir(), , "Répertoire Windows"
End Sub
```

Une chaîne de caractères C est une suite continguë de caractères terminée par un 0 binaire.

De plus, il faut que le programmeur ait déjà réservé une chaîne de caractères VBA avec suffisamment de place pour recevoir la chaîne de caractères C de la part de la fonction DLL appelée, sous peine de voir cette dernière n'importe où dans la mémoire de l'espace VBA !

Le concept de classe

- VBA intègre des concepts modernes tel que
  - classe
  - objet
  - masquage d'information

L'objet est le concept central en programmation orientée objet. C'est l'entité de base lors de l'exécution d'un programme.

Un objet occupe de la place en mémoire centrale et les données qu'il contient déterminent son état. Mais en plus de son état, il contient toutes les fonctions déterminant son comportement.

C'est ce qu'on appelle généralement l'encapsulation. Cette technique permet de garder ensemble et de considérer comme un tout les données et les fonctions modélisant une entité. L'encapsulation réduit donc la dispersion des données et des fonctions, et permet au programmeur de mieux appréhender et réduire la complexité d'un problème.

Bien que les informations soient toutes emmagasinées dans un objet, cela ne signifie nullement qu'elles soient toutes accessibles.

Le concept de classe

- encapsulation
- masquage d'information

L'ensemble des informations d'un objet est divisé en deux parties :

- une partie publique, visible et accessible en dehors de l'objet ;
- une partie privée, invisible et inaccessible en dehors de l'objet. Cette partie d'informations est bien sûr accessible et utilisable par les fonctions membres de l'objet.

C'est le principe du masquage d'information, très utile pour implanter de façon effective un niveau d'abstraction en cachant à l'utilisateur le « comment » de l'implémentation de l'objet.

Le masquage d'information permet aussi de créer par la suite du code plus facilement modifiable et maintenable. Comme les utilisateurs ne savent pas comment une opération est réalisée, mais uniquement ce qu'elle produit comme effet sur l'objet, il est possible de changer la réalisation de l'opération (généralement pour la rendre plus efficace) sans répercussions négatives pour les utilisateurs de l'objet.

Le concept de classe

- communication avec un objet
  - messages
  - méthodes

La communication avec un objet ne peut se faire qu'à travers son interface publique. Elle est réalisée en envoyant un message à l'objet, c'est-à-dire le nom d'une opération avec les arguments adéquats. Le comportement de l'objet est défini comme l'ensemble des messages auxquels il peut répondre.

Lors de la réception d'un message, il exécute ce qu'on appelle une méthode, ou plus simplement une fonction réalisant l'opération associée au message.

Le concept de classe

- une classe définit un ensemble potentiel d'objets
- c'est une description statique

Une classe définit un ensemble potentiel d'objets. Tout objet résulte, au moment de l'exécution du programme, de l'instanciation d'une classe existante. Une classe est quant à elle une description statique, figée dans le programme de l'ensemble potentiel d'objets instanciables.

On utilise souvent une métaphore pour faire comprendre la distinction entre l'objet et la classe : l'usine (la classe) fabrique des produits (les objets). Le programmeur a évidemment la possibilité de préciser la façon dont un objet doit être produit.

Dans le programme, on ne voit que des classes ; à l'exécution, il n'existe que des objets. Ce comportement est comparable à celui d'un type. En fait, définir une classe équivaut à définir un type nouveau.

Le concept de classe

- En VBA, le concept de classe (class module) est un outil de réutilisation de code.

Le programmeur peut insérer (créer) un ou plusieurs "class module" dans son projet.

Un module créé pour un projet peut être aussi être exporté sous forme de fichier ayant l'extension *cls*.

Un tel fichier peut alors être importé dans un autre projet : *réutilisation de code* !

Le concept de classe

- Exemple :
  - création de la classe Rationnel implémentant les nombres rationnels
  - interface :
    - méthodes :
      - init — initialisation d'un rationnel
      - add — addition de deux rationnels
      - affiche — affichage d'un rationnel
      - numerateur — accesseur et modificateur du numérateur
      - denominateur — accesseur et modificateur du dénominateur

Pour illustrer le concept de classe en VBA, nous créons la classe *Rationnel* à partir de laquelle nous pourrons instancier des objets par la suite. Ces objets ne seront manipulables par le programmeur qu'à travers leur interface publique composée des cinq méthodes (fonctions) publiques indiquées plus haut.

Le concept de classe

- en VBA :
  - le nom du module de classe est le nom de la classe, c'est-à-dire le nom du type

Le programmeur doit choisir avec soin le nom du module de classe. Ce nom sera en effet le nom du type des objets instanciés sur base de la définition de cette classe.

Nous adoptons la convention d'écrire le premier caractère de ce nom en majuscules.

Le concept de classe

```
Private pNumerateur As Integer
Private pDenominateur As Integer

Sub init(ByVal num As Integer, ByVal denom As Integer)
Sub affiche()
Sub add(ByVal r As Rationnel, r1 As Rationnel)
Public Property Get Numerateur() As Integer
Public Property Get Denominateur() As Integer
Public Property Let Numerateur(ByVal num As Integer)
Public Property Let Denominateur(ByVal denom As Integer)
Private Sub normaliser(num As Integer, denom As Integer)
Private Function pgcd(ByVal n1 As Integer, ByVal n2 As Integer) As
    Integer
```

Ce transparent illustre l'interface complète de la classe Rationnel. Elle comprend les procédures *init*, *affiche* et *add* sous une forme connue.

Par contre, la procédure *normaliser* ainsi que la fonction *pgcd* sont différentes en ce sens quels sont déclarées privées (*private*). Cela signifie que ces deux sous-programmes ne peuvent être utilisées que dans le ce module de classe. Ils sont inaccessibles en-dehors de ce dernier.

Par défaut, lorsque le programmeur omet de spécifier *public* ou *private*, le système considère qu'il s'agit d'une définition publique !

La classeRationnel définit deux données membres privées : *pNumerateur* et *pDenominateur* (par convention, nous utilisons le préfixe *p* pour indiquer qu'il s'agit d'une donnée privée).

Tout objet instancié par la suite à partir de la classe *Rationnel* contiendra ces deux champs, mais le programmeur ne pourra les accéder directement.

Il devra passer par les procédures prévues à cet effet : *Property Get* et *Property Let* (il existe encore une *Property Set*).

Property Let sert à changer la valeur d'une donnée membre de type autre qu'objet tandis que *Property Get* sert à accéder en lecture à une donnée membre. *Property Set* sert à changer la valeur d'une donnée membre de type objet.

Le concept de classe

```
Public Sub init(ByVal num As Integer, ByVal denom As Integer)
   Call normaliser(num, denom)
   Dim tmp As Integer
   tmp = pgcd(Abs(num), Abs(denom))
   pNumerateur = IIf(denom < 0, -num, num) / tmp
   pDenominateur = IIf(denom < 0, -denom / tmp, denom / tmp)
End Sub

Private Sub normaliser(num As Integer, denom As Integer)
   ' normaliser : seul le numérateur est éventuellement négatif
   num = IIf(denom < 0, -num, num)
   denom = IIf(denom < 0, -denom, denom)
End Sub
```

```
Private Function pgcd(ByVal n1 As Integer, ByVal n2 As Integer)
As Integer
    n1 = Abs(n1)
    n2 = Abs(n2)
    While n1 <> n2
        If n1 > n2 Then
            n1 = n1 - n2
        Else
            n2 = n2 - n1
        End If
    Wend
    pgcd = n1
End Function
```

La procédure *init* a le droit d'appeler les *pgcd* et *normaliser* qui sont définies PRIVATE.

L'utilisteur final de la classe Rationnel n'est pas autorisé à le faire !

Le concept de classe

```
Public Property Get Numerateur() As Integer
   Numerateur = pNumerateur
End Property

Public Property Get Denominateur() As Integer
   Denominateur = pDenominateur
End Property

Public Property Let Numerateur(ByVal num As Integer)
   pNumerateur = num
End Property

Public Property Let Denominateur(ByVal denom As Integer)
   pDenominateur = denom
End Property
```

Property Let/Set/Get sont des fonctions/procédures permettant de manipuler les données membres de la classe : ici *pNumerateur* et *pDenominateur*.

Le programmeur, utilisateur de la classe Rationnel ne connaîtra pas les noms des données membres. Il les manipule via les Property Numerateur et Denominateur !

Le concept de classe

```
Sub add(ByVal r As Rationnel, r1 As Rationnel)
   Dim num As Integer, denom As Integer
   num = (pNumerateur * r.Denominateur) + (r.Numerateur * pDenominateur)
   denom = pDenominateur * r.Denominateur
   Dim tmp As Integer
   tmp = pgcd(Abs(num), Abs(denom))
   num = num / tmp
   denom = denom / tmp
   Call normaliser(num, denom)
   r1.Numerateur = num
   r1.Denominateur = denom
End Sub

Sub affiche()
   Debug.Print pNumerateur; "/"; pDenominateur
End Sub
```

Les méthodes de la classe ont le droit d'accéder aux données membres sans passer par les *Property*.

L'objet exécutant la méthode *Add* additionne l'argument *r* "à soi-même" et renvoie le résultat à travers l'argument *r1*.

L'accès à une donnée membre publique ou à une fonction/procédure/property publique d'un objet s'écrit :

objet.donnée/fonction/procédure/property

Le concept de classe

```
Dim r1 As Rationnel, r2 As
  Rationnel
Dim r3 As Rationnel, r4 As
  Rationnel

Set r1 = New Rationnel
Set r2 = New Rationnel
r1.init 1, 3
r2.init 2, 5
r1.affiche          ' 1 / 3
r2.affiche          ' 2 / 5

Set r3 = New Rationnel
Call r1.add(r2, r3)
r3.affiche          ' 11 / 15
```

```
'   r1 = r3       ' ERREUR r1 et r3 sont des objets
    Set r1 = r3
    r1.affiche          ' 11 / 15

    Set r4 = New Rationnel
    Call r4.init(2, -4)
    r4.affiche          '- 1 / 2

    r1.Numerateur = 1: r1.Denominateur = 3
    r2.Numerateur = 2: r2.Denominateur = -4
    r1.add r2, r3
    r3.affiche          '- 1 / 6
```

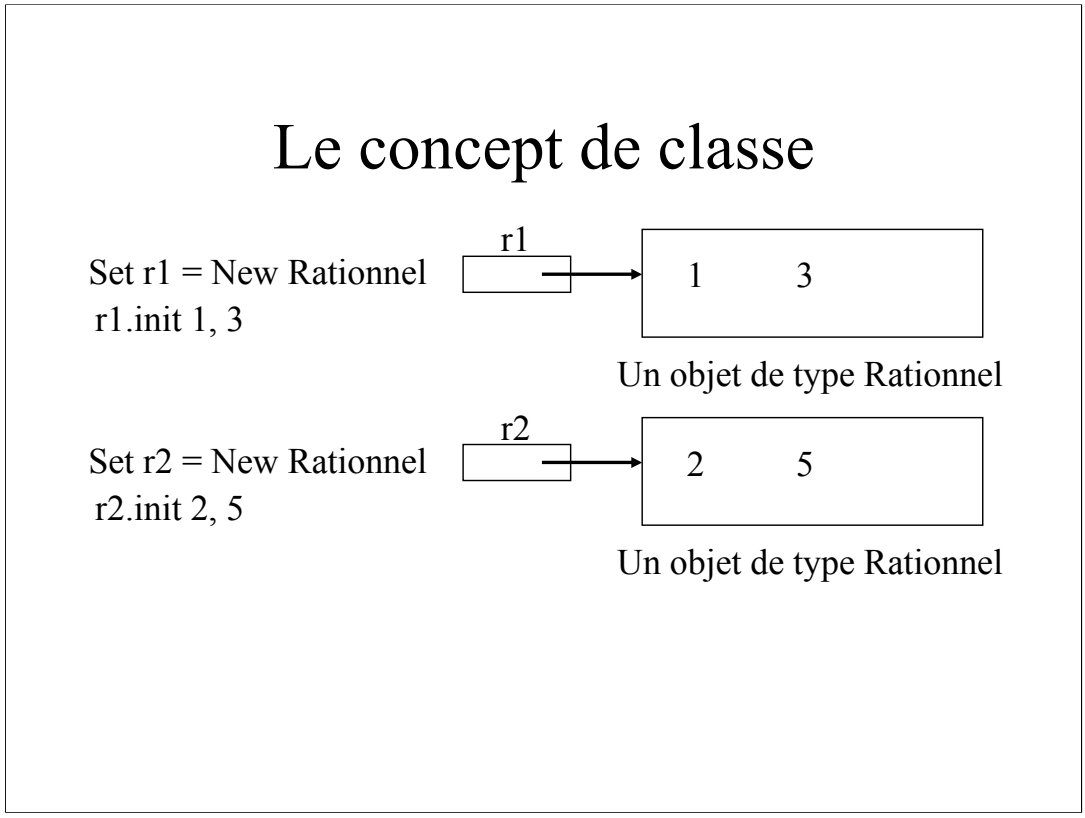
Pour l'utilisateur final de la classe *Rationnel*, *Rationnel* est un nouveau type.

De manière interne, une variable de type classe est en fait un pointeur sur un objet de cette classe. L'instruction DIM ne fait que de déclarer le pointeur et c'est à l'aide du mot-clé NEW que l'objet est créé dans la mémoire de l'ordinateur.

Pour affecter une nouvelle valeur à un objet, il faut obligatoirement utiliser l'instruction SET.

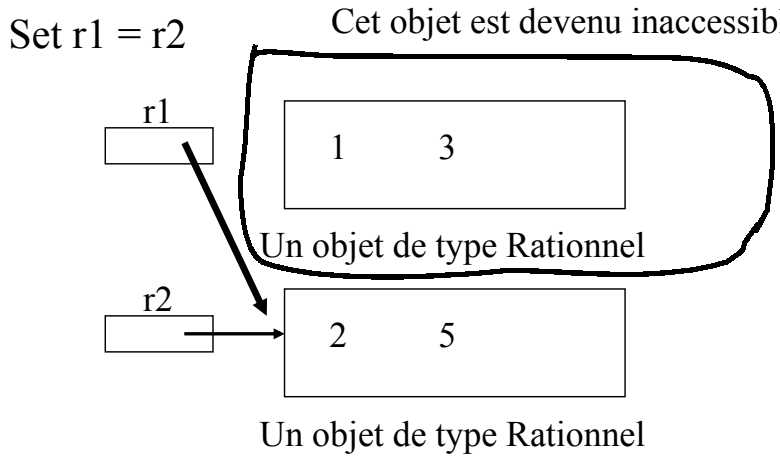
Comme le montre cet exemple, une fonction ou une procédure membre peut être appelée avec la syntaxe d'une fonction ou procédure normale.

Une property par contre s'utilise comme si c'était une donnée membre !



R1 et *r2* pointent en fait chacun sur un objet de type *Rationnel*. Dans le jargon informatique, on dit que ce sont des pointeurs.

Le concept de classe



En travaillant avec des pointeurs, il faut toujours faire attention à ne pas “perdre” involontairement l’objet sur lequel on pointe.

Lorsque l’on veut se débarasser explicitement d’un objet devenu inutile, il suffit de lui affecter la valeur ***nothing***.

Le concept de classe

```
Private Sub Class_Initialize()
   Numerateur = 0
   Denominateur = 1
   Debug.Print "Création du rationnel " & Me.Numerateur &
   "/" & Me.Denominateur
End Sub

Private Sub Class_Terminate()
   Debug.Print "Destruction du rationnel " & Me.Numerateur
   & "/" & Me.Denominateur
End Sub
```

A côté des fonctions et procédures applicables sur des objets, il existe deux procédures de classe, liées aux événements *Initialize* et *Terminate*.

Lorsqu'un objet est créé, automatiquement la procédure *Class_Initialize* est invoquée sans que le programmeur ait à le faire explicitement, sous réserve évidemment d'avoir défini une action pour cet événement.

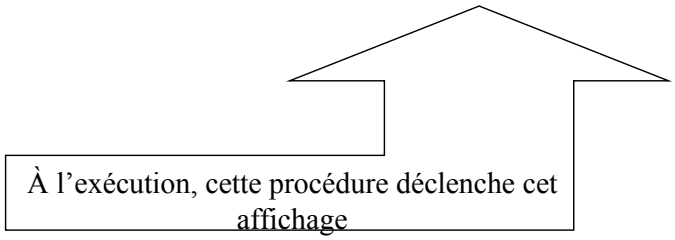
De même, lorsqu'un objet vient à mourir, c'est-à-dire qu'il est détruit parce qu'aucun pointeur ne pointe plus sur lui, automatiquement, la procédure de classe *Class_Terminate* est exécutée, sous réserve qu'elle a été définie.

Le concept de classe

```
Sub testNothing()
   Dim r1 As Rationnel
   Set r1 = New
Rationnel
   r1.affiche
   Set r1 = Nothing

   Set r1 = New
Rationnel
   r1.init 3, 8
   r1.affiche
End Sub
```

```
Création du rationnel 0/1
 0 / 1
Destruction du rationnel 0/1
Création du rationnel 0/1
 3 / 8
Destruction du rationnel 3/8
```



A côté des fonctions et procédures applicables sur des objets, il existe deux procédures de classe, liées aux événements *Initialize* et *Terminate*.

Lorsqu'un objet est créé, automatiquement la procédure *Class_Initialize* est invoquée sans que le programmeur ait à le faire explicitement, sous réserve évidemment d'avoir défini une action pour cet événement.

De même, lorsqu'un objet vient à mourir, c'est-à-dire qu'il est détruit parce qu'aucun pointeur ne pointe plus sur lui, automatiquement, la procédure de classe *Class_Terminate* est exécutée, sous réserve qu'elle a été définie.

