



idrissi_altf4@hotmail.com

Visual Basic

Introduction :

- Visual Basic (VB) est un langage de programmation évènementiel de troisième génération ainsi qu'un environnement de développement intégré, créé par Microsoft pour son modèle de programmation COM. Visual Basic est directement dérivé du BASIC et permet le développement rapide d'applications, la création d'interfaces utilisateur graphiques, l'accès aux bases de données en utilisant les technologies DAO, ADO et RDO, ainsi que la création de contrôles ou objets ActiveX. Les langages de Script tels que Visual Basic for Applications et VBScript sont syntaxiquement proches de Visual Basic, mais s'utilisent et se comportent de façon sensiblement différente.

- Visual Basic est un des langages les plus utilisés pour l'écriture d'applications commerciales. Dans une étude conduite en 2005, 62 pourcent des développeurs déclaraient utiliser l'une ou l'autre forme de Visual Basic. Actuellement, les langages les plus utilisés dans le domaine commercial sont Visual Basic, C++, C#, Java. JavaScript est également un concurrent de Visual Basic.

Visual Basic, un langage (presque) objet

- En quoi un langage objet diffère-t-il d'un langage normal ? On peut résumer l'affaire en disant qu'un langage objet possède toutes les caractéristiques d'un langage traditionnel, avec deux grands aspects supplémentaires.
- Donc, c'est un premier point, on peut tout à fait programmer dans un langage objet comme on programmerait du Fortran, du Cobol ou du C. Selon le vieil adage, qui peut le plus peut le moins. En pratique, cela voudrait dire négliger tout ce qui fait la spécificité d'un tel langage, comme - entre autres - la prise en charge de l'environnement graphique Windows.

Visual Basic, un langage (presque) objet

- Cela implique également que toutes les notions fondamentales que le programmeur a mises en pratique en algorithmique ou en programmation dans un langage traditionnel conservent leur validité pleine et entière : comme tout langage, un langage objet ne connaît que quatre grands types d'instructions : affectations de variables, tests, boucles et entrées / sorties (encore que, nous le verrons, ce dernier type puisse y connaître de fait un certain nombre de bouleversements). Comme tout langage, un langage objet connaît des variables de différents types (numérique, caractère, booléen), et des variables indicées (tableaux). Donc, encore une fois, tout ce qui était vrai dans la programmation traditionnelle demeure vrai dans la programmation objet.
- Mais celle-ci offre comme on vient de le dire deux nouveaux outils, redoutables de puissance, à la trousse du programmeur.

Particularités des langages objet

- En quoi un langage objet diffère-t-il d'un langage normal ? On peut résumer l'affaire en disant qu'un langage objet possède toutes les caractéristiques d'un langage traditionnel, avec deux grands aspects supplémentaires.
- Donc, c'est un premier point, on peut tout à fait programmer dans un langage objet comme on programmerait du Fortran, du Cobol ou du C. Selon le vieil adage, qui peut le plus peut le moins. En pratique, cela voudrait dire négliger tout ce qui fait la spécificité d'un tel langage, comme - entre autres - la prise en charge de l'environnement graphique Windows.
- Cela implique également que toutes les notions fondamentales que le programmeur a mises en pratique en algorithmique ou en programmation dans un langage traditionnel conservent leur validité pleine et entière : comme tout langage, un langage objet ne connaît que quatre grands types d'instructions : affectations de variables, tests, boucles et entrées / sorties (encore que, nous le verrons, ce dernier type puisse y connaître de fait un certain nombre de bouleversements). Comme tout langage, un langage objet connaît des variables de différents types (numérique, caractère, booléen), et des variables indicées (tableaux). Donc, encore une fois, tout ce qui était vrai dans la programmation traditionnelle demeure vrai dans la programmation objet.

Les Objets

- La première particularité d'un langage objet est de mettre à votre disposition des objets.
- Un objet peut être considéré comme une structure supplémentaire d'information, une espèce de super-variable. En effet, nous savons qu'une variable est un emplacement en mémoire vive, caractérisé par une adresse – un nom – et un type (entier, réel, caractère, booléen, etc.). Dans une variable, on ne peut stocker qu'une information et une seule. Même dans le cas où l'on emploie une variable indicée – un tableau – les différents emplacements mémoire ainsi définis stockeront tous obligatoirement des informations de même type.
- Un objet est un groupe de variables de différents types. Il rassemble ainsi couramment des dizaines d'informations très différentes les unes des autres au sein d'une même structure, rendant ainsi ces informations plus faciles à manier.
- A la différence de ce qui se passe avec un tableau, les différentes variables d'un même objet ne sont pas désignées par un indice, mais par un nom qui leur est propre. En l'occurrence, ces noms qui caractérisent les différentes variables au sein d'un objet s'appellent des propriétés de l'objet. Conséquence, toute propriété d'objet obéit strictement aux règles qui s'appliquent aux variables dans tout langage (type, taille, règles d'affectation...).

Les Objets

- On dira également que plusieurs objets qui possèdent les mêmes propriétés sont du même type, ou encore pour mieux frimer, de la même classe.
- A titre d'exemple, prenons un objet d'usage courant : un ministre.
- Les propriétés d'un ministre sont : sa taille, son poids, son âge, son portefeuille, le montant de son compte en Suisse, son nom, sa situation par rapport à la justice, etc.
- On peut retrouver aisément le type de chacune de ces propriétés :
- le portefeuille, le nom, sont des propriétés de type caractère.
- la taille, le poids, l'âge, le compte en Suisse, sont des propriétés de type numérique.
- la situation judiciaire (mis en examen ou non) est une propriété booléenne.

Syntaxe

- La syntaxe qui permet de désigner une propriété d'un objet est :
- `objet.propriété`
- Par exemple, nous pouvons décider que le montant du compte en Suisse du ministre Duchemol s'élève modestement à 100 000 euros. Si la propriété désignant ce compte pour les objets de type (de classe) `ministre` est la propriété `CompteSuisse`, on écrira donc l'instruction suivante :
- `Duchemol.CompteSuisse = 100 000`
- Pour affecter à la variable `Toto` le montant actuel du compte en Suisse du ministre Duchemol, on écrira :
- `Toto = Duchemol.CompteSuisse`
- Pour augmenter de 10 000 euros le montant du compte en Suisse de Duchemol, on écrira :
- `Duchemol.CompteSuisse = Duchemol.CompteSuisse + 10 000`
- Et, vraiment juste histoire d'utiliser une propriété booléenne, et parce que Duchemol n'est pas le seul objet de la classe `ministre` :
- `Pasqua.MisEnExamen = True`
- On répète donc qu'hormis ce qui concerne la syntaxe, l'usage des propriétés des objets ne se différencie en rien de celui des variables classiques.

Méthodes

- Les langages objet ont intégré une autre manière d'agir sur les objets : les méthodes.
- Une méthode est une action sur l'une – ou plusieurs - des propriétés d'un objet.
- Une méthode va supposer l'emploi d'un certain nombre d'arguments, tout comme une fonction. On trouvera donc des méthodes à un argument, des méthodes à deux arguments (plus rares), et aussi des méthodes sans arguments.
- Ce n'est pas le seul point commun entre les méthodes et les fonctions.
- On sait qu'une fonction peut : soit accomplir une tâche impossible si elle n'existait pas, soit accomplir une tâche possible par d'autres moyens, mais pénible à mettre en œuvre.
- De la même manière, certaines méthodes accomplissent des tâches qui leur sont propres, et qui ne pourraient pas être accomplies si elles n'existaient pas. D'autres méthodes ne sont là que pour soulager le programmeur, en permettant de modifier rapidement un certain nombre de propriétés.

Méthodes

- Par exemple, reprenons le cas notre ministre. Une méthode pourrait être AugmenterPatrimoine, qui supposerait un argument de type numérique. On pourrait ainsi écrire :
- Duchemol.AugmenterPatrimoine(10 000)
- Ce qui aurait en l'occurrence exactement le même effet que de passer par la propriété correspondante :
- Duchemol.CompteSuisse = Duchemol.CompteSuisse + 10 000

Conclusion

- Pour terminer sur ce sujet, il faut bien faire attention à une chose lorsqu'on utilise des objets.
- certains objets sont fournis par le langage de programmation lui-même. Il s'agit en particulier (mais pas seulement) de ce qu'on appelle des contrôles, c'est-à-dire d'objets possédant pour la plupart une existence graphique ; ce sont des éléments de l'interface Windows. Pour tous ces objets que le programmeur utilise alors qu'ils ont été créés par d'autres, les propriétés et les méthodes ne s'inventent pas : chaque type (chaque classe) d'objet possède ses propres méthodes et arguments, qu'il s'agit donc de connaître pour utiliser l'objet en question.
- d'autre part, un langage objet ouvre la possibilité de créer soi-même ses propres objets, et donc de programmer leurs propriétés et leurs méthodes. On se situe alors à tout autre niveau, celui de la programmation objet proprement dite.

Procédures événementielles

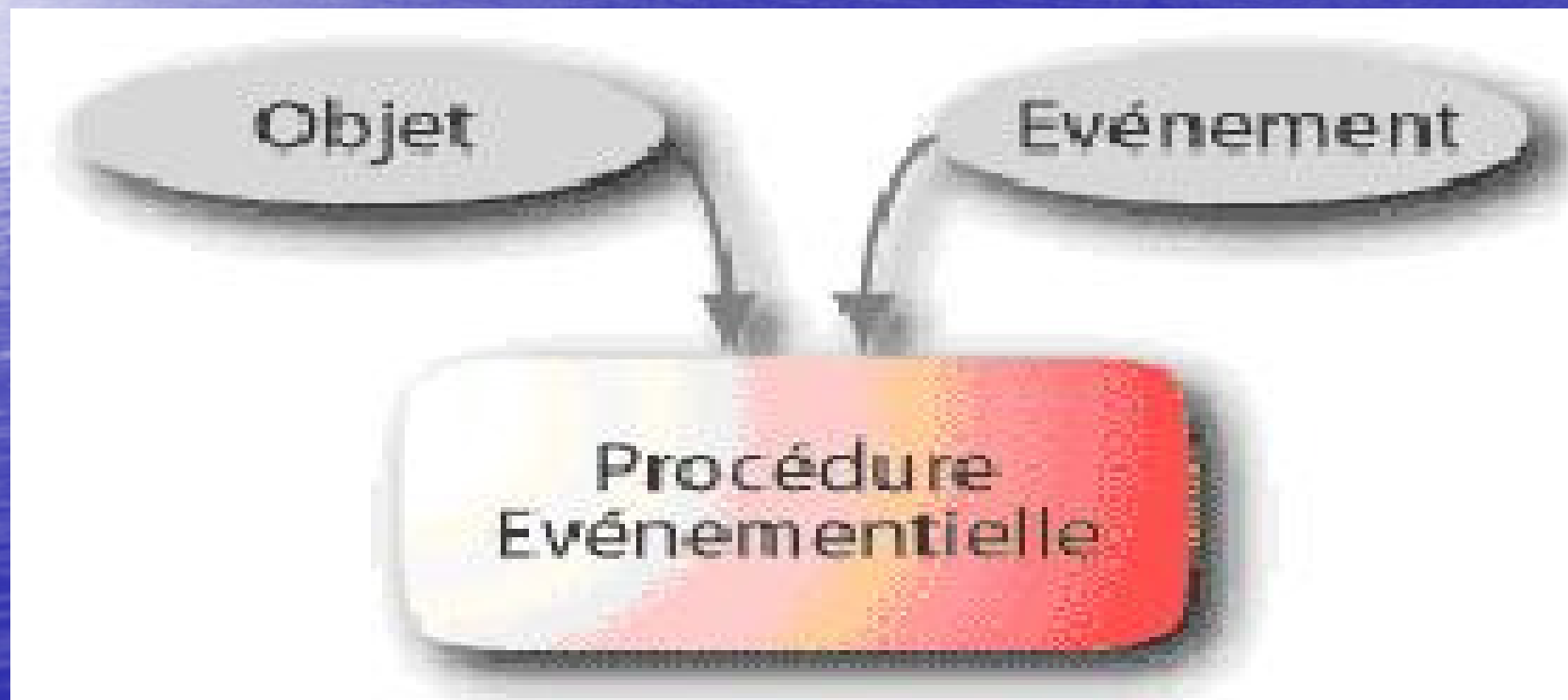
- On en arrive à la deuxième grande possibilité supplémentaire des langages objet par rapport aux langages traditionnels.
- En PASCAL ou en C, par exemple, une application est constituée d'une procédure principale contenant la totalité du code (y compris par l'appel indirect à des sous-programmes). Les instructions qu'elle contient sont exécutées les unes après les autres, jusqu'à la fin (je passe pudiquement sous silence l'improbable hypothèse d'un arrêt prématuré pour cause d'erreur).
- Le point fondamental est que dans un tel langage, l'ordre d'exécution des procédures et des sous-procédures est entièrement fixé d'avance par le programmeur lui-même, par le biais des instructions d'appel des sous-procédures. Par ailleurs, et ce n'est pas un hasard, ces procédures portent des noms arbitraires fixés par le programmeur, hormis le cas particulier de la procédure principale, qui se doit de porter un nom particulier, fixé par le langage (généralement, Main).
- Dans un langage objet, on peut, si on le désire, conserver intégralement ce mode de fonctionnement. Mais ce n'est plus le seul possible.
- En effet, dans un langage objet, il n'y a donc plus à proprement parler de procédure principale ; en tout cas, l'existence d'une procédure principale n'a rien d'obligatoire.

Procédures événementielles

- Chaque procédure est liée à la survenue d'un événement sur un objet, et sera donc automatiquement exécutée lorsque cet événement se produit. Le nom de la procédure est alors, de manière obligatoire, le nom de la combinaison objet-événement qui la déclenche.
- On vient de parler des objets, et en particulier des contrôles. Répétons qu'un contrôle est un des éléments de l'interface graphique de Windows, éléments que VB met à la disposition du programmeur pour qu'il constitue ses propres applications. Ainsi, les contrôles les plus fréquents sont : la feuille, le bouton de commande, la liste, la case à cocher, le bouton radio, etc.
- Quant aux événements, ils peuvent être déclenchés par l'utilisateur, ou par déroulement du programme lui-même. Les événements déclenchés par l'utilisateur sont typiquement : la frappe au clavier, le clic, le double-clic, le cliquer-glisser. Les événements déclenchés par le code sont des instructions qui modifient, lors de leur exécution, une caractéristique de l'objet ; par exemple, le redimensionnement, le déplacement, etc.

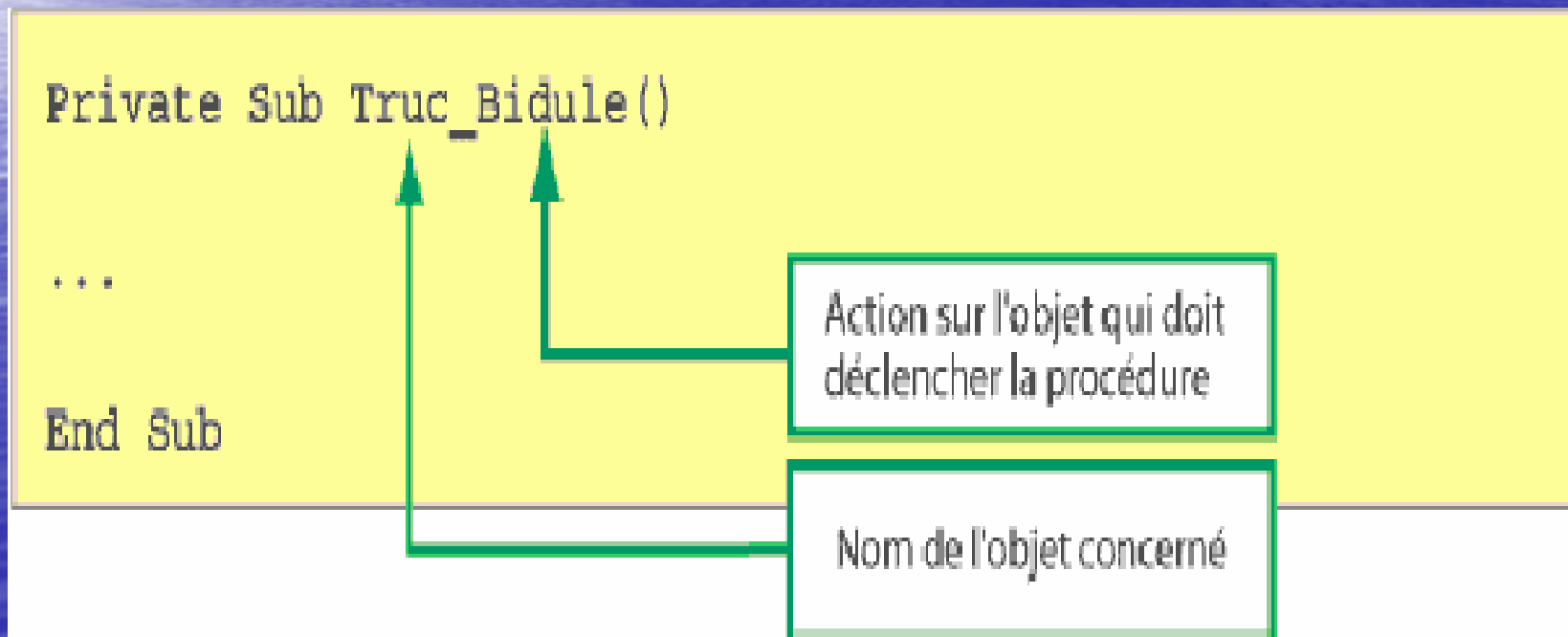
Procédures événementielles

- Résumons-nous. Un petit dessin vaut parfois mieux qu'un grand discours :



Procédures événementielles

- Le lien entre l'objet, la survenue de l'événement et le déclenchement de la procédure est établi par le nom de la procédure lui-même. Ainsi, le nom d'une procédure événementielle répond à une syntaxe très précise :



Procédures événementielles

- Par exemple, la procédure suivante :
- Private Sub Machin_Click()
...
End Sub
- Se déclenche si et seulement si l'utilisateur clique sur l'objet dont le nom est "Machin". Si on prend le problème dans l'autre sens : si je suis programmeur, et que je veux qu'il se passe ceci et cela lorsque l'utilisateur clique sur le bouton appelé "Go", je dois créer une procédure qui s'appellera obligatoirement Sub Go_Click() et qui contiendra les instructions "ceci" et "cela".
- A la différence d'un programme traditionnel, les procédures liées aux différents événements possibles ne seront donc pas toujours exécutées dans le même ordre : tout dépend de ce que fera l'utilisateur.
- Mais bien sûr, à l'intérieur de chaque procédure, les règles traditionnelles de programmation restent vraies.

Compilation et Interprétation

- Lorsqu'on écrit une application Visual Basic (on ne traite ici que du cas standard, il en existe d'autres, mais qui sortent du sujet du cours), on crée donc un ensemble d'objets à partir des classes proposées, et on définit les procédures qui se rapportent à ces objets. Lorsqu'on sauvegarde cette application, Visual Basic va créer un certain nombre de fichiers. Pour l'essentiel :
 - un fichier dit Projet comportant l'extension *.vbp (les plus fûtés d'entre vous reconnaîtront là l'acronyme de *Visual Basic Project*. Ils sont décidément très forts, chez Microsoft). Ce fichier rassemble les informations générales de votre application (en gros, la structure des différents objets Form, qui sont le squelette de toute application).
 - un fichier par objet Form créé, fichier portant l'extension *.frm. Ne soyez pas si impatients, vous saurez très bientôt ce qu'est un objet Form. Toujours est-il que si votre application comporte six Form, vous aurez en plus du fichier "projet", six fichiers "Form" à sauvegarder. Chacun de ces fichiers comporte les objets contenus par la "Form", ainsi que tout le code des procédures liées à ces objets.

Compilation et Interprétation

éventuellement, d'autres fichiers correspondant à d'autres éléments de l'application, éléments dont nous parlerons plus tard (modules, modules de classe.

- La destruction de l'un quelconque de ces fichiers vous portera naturellement un préjudice que l'on ne saurait sous-estimer.
- D'autre part, je tiens à signaler dès maintenant qu'il est extrêmement périlleux de procéder à des "copier – coller" de Form, car le fichier structure (vbp) possède une tendance affirmée à se mélanger complètement les crayons en pareil cas.
- Conclusion, on crée un nouveau projet à chaque nouvelle application, et on ne déroge jamais à cette règle d'or. Il ne doit jamais y avoir deux projets ouverts en même temps dans la même fenêtre VB, sous peine de graves représailles de la part du logiciel.

Compilation et Interprétation

- Tant que votre projet est ouvert sous cette forme d'une collection de fichiers vbp et frm, vous pouvez naturellement l'exécuter afin de le tester. Lors de l'exécution, le langage est alors ce qu'on appelle « compilé à la volée ». C'est-à-dire que VB traduit vos lignes de code au fur et à mesure en langage machine, puis les exécute. Cela ralentit naturellement considérablement l'exécution, même si sur de petites applications, c'est imperceptible. Mais dès que ça commence à grossir...
- Voilà pourquoi, une fois l'application (le "projet") mis au point définitivement, VB vous propose de le compiler une bonne fois pour toutes, créant ainsi un unique fichier *.exe. Ce fichier contient cette fois à lui seul l'ensemble de votre projet, form, code,. Et il peut naturellement être exécuté sans l'ouverture – donc la possession - préalable de Visual Basic (à un détail près, que nous réexaminerons plus loin dans ce cours).
- Un projet terminé est donc un projet compilé.
- Et qui, accessoirement, fonctionne sans erreurs.

L'interface VB

- le langage possède un vérificateur de syntaxe en temps réel. C'est-à-dire que l'éditeur de code détecte les instructions non légitimes au fur et à mesure que vous les entrez, et il les signale par un texte mis en rouge, et pour faire bonne mesure, par un message d'erreur. Il est donc impossible de laisser traîner des erreurs de syntaxe en VB, ce qui je n'en doute pas, rendra les amateurs de langage C inconsolables. Qu'ils se rassurent toutefois, VB ne corrige ni les fautes de logique, ni les fautes fonctionnelles. Il reste donc tout de même de quoi se faire plaisir.
- De même, le code est immédiatement mis en couleurs par l'éditeur.
- le bleu correspond aux mots réservés du langage (instructions, mots-clés...)
- le vert correspond à un commentaire (toute ligne commençant par un guillemet simple – quote – est considérée comme un commentaire).
- tout mot reconnu par l'éditeur (nom d'objet, instruction) voit sa première lettre transformée automatiquement en majuscule.

L'interface VB

- tout nom d'objet suivi d'un point voit s'afficher une liste déroulante contenant l'intégralité des propriétés et des méthodes disponibles pour cet objet. A contrario, cela signifie qu'un objet ne faisant pas apparaître une liste déroulante dans le code est un objet non reconnu (qui n'existe pas). De même, une instruction ne prenant pas automatiquement de majuscule initiale est une instruction non reconnue.
- il est possible de réaliser une exécution pas à pas via la commande appropriée du menu.
- il est possible d'insérer des points d'arrêt pour faciliter le débogage.
- dans le cas d'un pas à pas comme d'un point d'arrêt, il est possible de connaître la valeur actuelle d'une variable en pointant (sans cliquer !) la souris sur une occurrence de cette variable dans le code.

Fenêtre d'une application VB en exécution Pas à Pas

le pointeur de la souris, en survolant i, fait apparaître sa valeur actuelle

stade actuel de l'exécution pas à pas

point d'arrêt

```
Project1 - Form1 (Code)
Command1 Click
Private Sub Command1_Click()
    For i = 1 To 12
        i = 5: j = 5 To 9
        ' on fait ce qu'on veut.
    Next j
Next i
End Sub
```

commentaire

Partie 1

Premiers Eléments du Code

- Visual Basic s'est voulu un langage simple. Vous pourrez juger par vous-mêmes de la réalité de cette prétention, mais il faut reconnaître que pour un certain nombre de choses élémentaires, les développeurs de Microsoft ont choisi de ne pas compliquer inutilement la vie du programmeur.
- Nous allons donc effectuer un premier, et très rapide, tour d'horizon, pour découvrir les syntaxes de base de ce langage.

Variables

- Pour ce qui est des noms de variables, VB ne fait que dans le très classique. Voyez donc :
- Les noms de variables n'ont pas de longueur maximale
- Ils doivent commencer par un caractère
- Ils ne doivent pas comporter d'espace
- Ils ne sont pas sensibles à la casse (Toto et toto sont la même variable)
- En ce qui concerne la déclaration de ces variables, celle-ci est optionnelle (en tout cas pour les variables locales, on reparlera dans un instant de cet aspect). Cela signifie que dans l'option par défaut, il n'est pas besoin de déclarer les variables pour s'en servir. Dans le cadre d'une grosse application, on devra activer l'option de déclaration des variables, ce qui permettra de gagner de l'espace mémoire (car les variables se déclarent par défaut avec le type le plus gourmand en mémoire, on y reviendra).
- Pour les types des variables, c'est tout pareil, rien que du très ordinaire et sans aucune surprise

Variables

- Boolean : True – False
- Byte : de 0 à 255
- Integer : de -32 768 à 32 767
- Long : de -2 à +2 milliards environ
- Single : virgule flottante simple précision
- Double : virgule flottante double précision
- Currency : entier en virgule fixe
- Enfin, l'instruction d'affectation est le signe égal (=).

Déclaration, or not déclaration ?

- On vient de voir que VB offrait la possibilité de déclarer ou non les variables (en tout cas, les variables locales). Mais, dans l'hypothèse de non déclaration systématique des variables, nous avons pudiquement glissé sur ce qui se passe lorsqu'une variable non déclarée est utilisée...
- En fait, cette variable est automatiquement créée dans un type spécial : le type Variant. Ce type possède comme particularité d'être "souple", à savoir de pouvoir s'ajuster à n'importe quel contenu (caractère, booléen, n'importe quel sous-type de numérique). Mais, évidemment, tout à un prix, et s'il est très pratique, le type variant n'en demeure pas moins extrêmement gourmand en mémoire vive occupée.
- Donc, si dans le cadre d'un petit exercice, on peut se permettre de ne pas déclarer ses variables, ce n'est pas le cas dans un projet, où l'accumulation des variables de type Variant peut conduire à de graves dysfonctionnements.

Portée des variables

- L'existence d'une variable peut se dérouler sur trois niveaux :
- Niveau Procédure : cela veut dire que la variable est locale. Dès que l'on quitte la procédure en question, la variable disparaît, et son contenu avec elle. Pour déclarer une variable au niveau procédure, on tape au sein de la procédure considérée
- Dim NomVariable as Type
- Niveau Form : la variable est disponible pour toutes les procédures de la Form , mais pas pour les procédures se situant sur une autre Form. Pour déclarer une variable au niveau Form, on tape tout en haut de la Form, à l'extérieur des procédures :
- Dim NomVariable as Type
- Non, ce n'est pas une erreur, c'est bien la même chose que précédemment. Mais l'emplacement de l'instruction n'est pas le même, et c'est cela qui change tout.

Portée des variables

- Niveau Projet : la variable est disponible, et sa valeur est conservée pour toutes les procédures de l'application, quel que soit leur emplacement. Pour déclarer une variable globale, il faut d'abord créer un module. Un module est un type de feuille destiné uniquement à recevoir du code, et qui n'a donc pas, contrairement aux Form, d'apparence graphique. C'est dans un module qu'on écrit la procédure principale, lorsqu'on en veut une, et qui de là pilote les différentes Form, elles mêmes donnant accès aux procédures liés aux objets qu'elles contiennent. Sur ce module, donc, on écrit :
 - `Public NomVariable as Type`
 - Naturellement, est-il besoin de le préciser, il ne faut pas raisonner en termes d'artillerie lourde, et déclarer toutes les variables au niveau projet, en se disant que comme ça, on est blindé : car ce blindage, par l'excès de place mémoire, ralentira votre application, au besoin considérablement. Il faut donc pour chaque variable se demander à quel niveau on en a besoin, et faire les déclarations en fonction.

Variables indicées

- On appelle aussi cela des tableaux ! Ce peut être des tableaux de nombres, de chaînes, de booléens, bref, de tout ce qu'on veut. Quant on crée un tableau, soit on sait d'avance combien d'éléments il va englober, soit on veut qu'il soit dynamique (mais cela se paye bien sûr par une perte de rapidité à l'exécution). Tout tableau doit obligatoirement être déclaré, quel que soit par ailleurs le réglage de l'option de déclaration des variables.
- Pour créer un tableau de 12 entiers, on écrira :
- `Dim MonTableau(11) As Integer`
- Pour créer un tableau dynamique on écrira :
- `Dim MonTableau() As Integer`
- Ensuite, dès qu'on veut en fixer la taille, on écrit dans le code :
- `Redim MonTableau(11)`
- Si ce redimensionnement doit s'effectuer en conservant les valeurs précédemment entrées dans le tableau, on ajoutera le mot-clé `Preserve` :
- `Redim Preserve MonTableau(11)`
- En résumé, aucun problème spécifique ici que vous n'ayez déjà abordé sur un langage précédent (le C, par exemple, à tout hasard).

Opérateurs

- C'est d'une monotonie désespérante, mais là non plus, il n'y a aucune surprise en ce qui concerne les opérateurs, qui sont absolument standard. Je les rappelle rapidement :
- Opérateurs numériques : + - * /
- Opérateurs booléens : And Or Xor Not
- Opérateur caractères : & (concaténation)

Tests

- Structures possibles :

- If ... Then

...
EndIf

- If ... Then

...
Else

...
EndIf

- If ... Then

...
Elseif ... Then

...
Elseif ... Then

...
Else

...
EndIf

Boucles

- Aucun problème. Il y a en fait abondance de biens, car VB propose une liste interminable de possibilités. Parmi celles-ci, on retrouve les deux fondamentaux précédemment étudiés dans le susdit extraordinaire [cours d'algorithmique](#) :
- While ...
...
Wend
- For x = a to b Step c
...
Next x

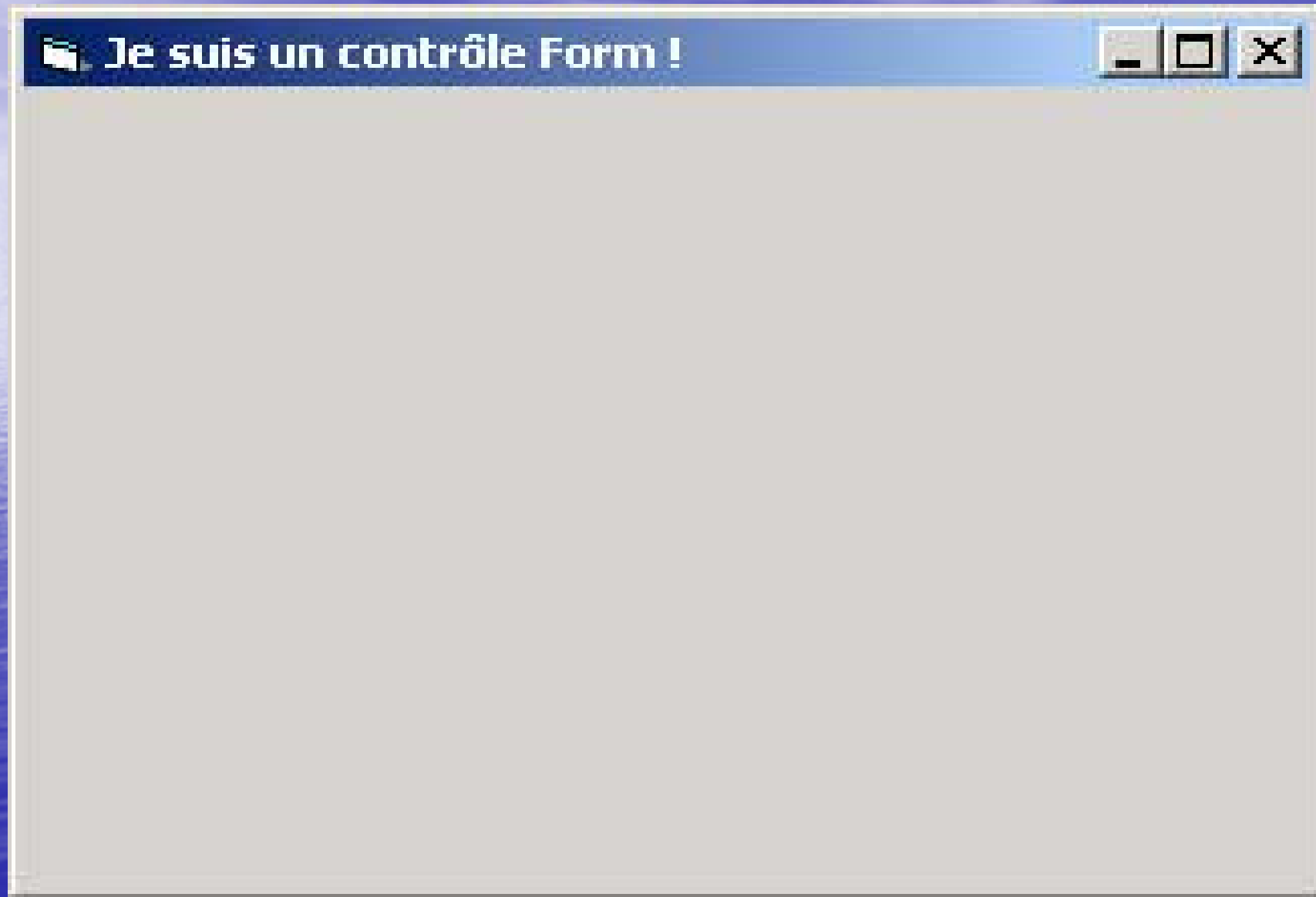
Partie 2

Les Premiers Contrôles

Le contrôle Form

- Le contrôle de base, absolument universel, en Visual Basic est la feuille, ou formulaire, en anglais, Form. Cet objet est incontournable ; on ne peut créer et utiliser d'autres objets que si ceux-ci font partie d'une Form. Il y a certes moyen de rendre celle-ci invisible, pour qu'on ne voie que les objets posés dessus ; mais même invisible, elle doit nécessairement exister.
- A l'ouverture de Visual Basic, on vous propose d'ailleurs par défaut un objet Form. Ce n'est pas pour rien.

1. Le contrôle Form



Le contrôle Form

- Nous allons examiner d'emblée deux propriétés dont il est essentiel de comprendre la signification. Ces propriétés existent pour l'objet Form, mais elles existent aussi pour tous les autres objets Visual Basic. Ce que vous allez apprendre maintenant est donc d'une utilité universelle.
- A tout Form en particulier, et à tout contrôle en général, sont donc associées les propriétés :
- Name : il s'agit du nom de l'objet tel qu'il est géré par l'application. Cela correspond en quelque sorte à un nom de variable (sauf que ce n'est pas une variable, c'est un objet !). Par défaut, VB baptise tous les objets que vous créez de très jolis noms génériques, comme Form1, Form2, Form3, Text1, Text2, Text3, etc. Ainsi, dès que l'application devient importante, il est très vivement conseillé de rebaptiser les objets dès leur création, afin de leur donner des noms plus évocateurs. Parce que sinon, dès que le nombre d'objets va grossir, vous serez complètement paumés, et vous ne saurez plus qui est quoi.

Le contrôle Form

- Caption : il s'agit du texte associé à l'objet sur l'écran. Cette Caption est donc très utile pour professionnaliser une application, lui donner un look fini, mais ne joue aucun rôle dans la désignation de l'objet par l'application. C'est un rôle purement décoratif !
Dans le code, on ne désigne donc jamais un objet par sa Caption, mais par son Name. Dans le cas de la Form, par exemple, la propriété Caption désigne le texte qui vient s'écrire dans la barre de titre.
Et je rappelle que dans le cas particulier d'une Form, outre ces deux « noms », il y a en plus le nom du fichier dans lequel elle sera sauvegardée. Cela fait donc trois « noms » pour un seul contrôle, noms qui chacun jouent un rôle différent ; Attention à ne pas se mélanger les pinceaux !

Le contrôle Form

- Il y a des tas d'autres propriétés intéressantes de Form. Je ne vais bien sûr pas toutes vous les infliger. Il y a une aide, après tout, il faut que cela serve. Je signale toutefois :
- Modal : propriété qui gère le caractère « impératif » ou non de toute Form (on peut cliquer ou non en dehors de la form, sur une autre Form)
- Style : propriété qui gère la tête générale de la Form (présence ou non du menu système, tête des bordures, etc.)
- Visible : propriété booléenne qui gère, comme son nom l'indique, le caractère visible ou non de la Form (et partant, des autres contrôles qu'elle contient)

Le contrôle Form

- Rappel : procédures événementielles
- Je rappelle qu'à chaque objet créé peuvent correspondre autant de procédures que d'événements possibles survenant sur cet objet. En pratique, il est rare que l'on doive prévoir tous les événements. Si un événement se produit pour un objet (ex : l'utilisateur clique dessus) et qu'aucune procédure n'est attachée à cet événement, c'est très simple : il ne se passe rien !
- VB gère très simplement la création – ou la modification – d'une procédure liée à un objet. Lorsque l'objet est sélectionné, il suffit de faire un double-clic dessus (ou, c'est équivalent, d'appuyer sur F7). On se retrouve alors dans la fenêtre Code. Si elle n'existait pas encore, la procédure gérant l'événement le plus courant pour l'objet considéré vient d'être créée. Si elle existait déjà, cette manipulation vous amène sur elle pour modifications éventuelles.
- La procédure ainsi créée, je le rappelle, se présente ainsi :
- Private Sub NomObjet_Eventement()

```
...  
End Sub
```

1. Le contrôle Form

- Private signifie que la procédure n'est utilisable que pour la Form considérée (et pas pour d'autres objets situés sur d'autres Form). Vous en concluez que l'on peut aussi créer des procédures Public même si nous ne nous amuserons guère à cela dans le cadre de ce cours.
- End Sub marque la fin de la procédure en question (mais pas la fin de l'application).
- Vous remarquerez que l'événement retenu par VB comme étant le plus probable pour une Form est Load (chargement). Autrement dit, cette procédure contient tout ce que l'application doit faire au moment où la Form en question est chargée en mémoire vive.

Le CommandButton (Bouton de Commande)

- Il s'agit du bouton type OK, Annuler, mais dont le texte apparent (en Anglais, Caption) et le rôle dans une application peuvent varier à l'infini. Je ne vous étonnerai pas en vous disant que l'action que VB considère comme étant la plus commune pour les boutons de commande est Click (en Français, clic, note du traducteur, 20/20 au TOEIC).
- Quelques propriétés intéressantes de la classe CommandButton :
- Name : bien sûr !
- Caption : évidemment...
- Visible : ce qui ne surprend pas.
- Enabled : cette propriété, booléenne, est comme on le verra très loin d'être l'apanage des seuls boutons de commande. Elle permet (valeur True) à un contrôle d'être actif, c'est-à-dire de pouvoir recevoir des événements, et donc de déclencher des procédures. Inversement, elle interdit (valeur False) à un contrôle de recevoir quelque événement que ce soit de la part de l'utilisateur. Dans ce cas, le contrôle apparaît grisé à l'écran. Ça vous rappelle des choses déjà vécues sous Windows ? Normal, c'est absolument fait exprès.
- Style : si cette propriété possède comme valeur Graphical, il devient alors possible d'ajouter une image sur le bouton, ou d'en modifier la couleur. Dans le cas contraire (valeur Standard, par défaut), le bouton conserve l'aspect par défaut des boutons de Windows, à savoir un gris souris du meilleur goût.

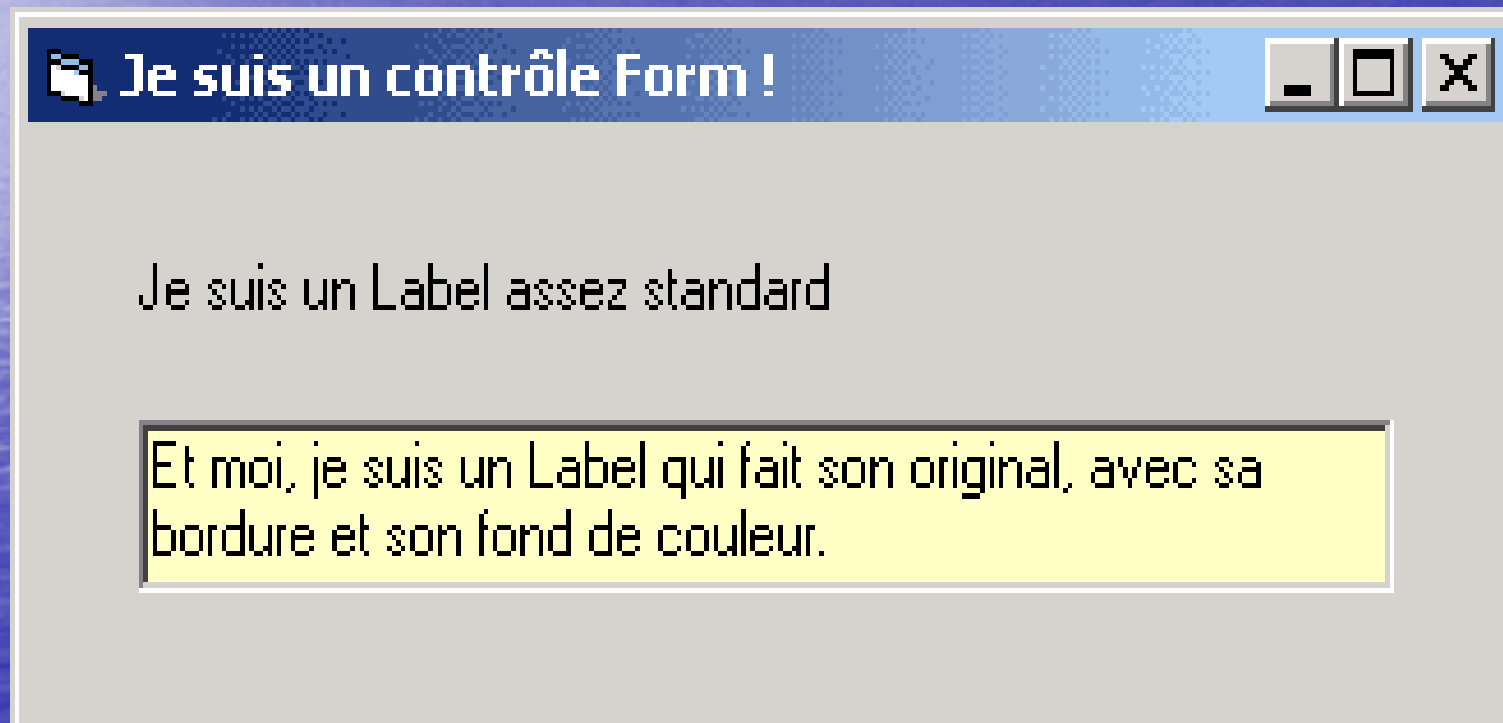
Le CommandButton (Bouton de Commande)

- Il est maintenant grand temps pour vous de passer aux premières joies de la programmation VB...
- Voici donc vos premiers exercices VB. Pour chacun d'eux, vous devez commencer par télécharger l'exécutable. C'est le modèle que vous devez copier au plus près. N'hésitez pas à passer un peu de temps à bien comprendre ce qu'il fait, avant de vous ruer sur votre clavier !

Pour l'**Application Micro**, il y a deux manières bien différentes de programmer ce que l'on voit à l'écran. L'une consiste à préparer deux objets, puis en cacher un et à faire apparaître l'autre. L'autre technique emploie un seul objet, dont elle modifie les caractéristiques. Les deux sources proposées correspondent à ces deux choix de programmation.

Le Label (Etiquette)

- Un Label est un contrôle "inerte", qui sert à afficher un texte sur une Form. Son aspect peut varier quelque peu selon les styles adoptés :

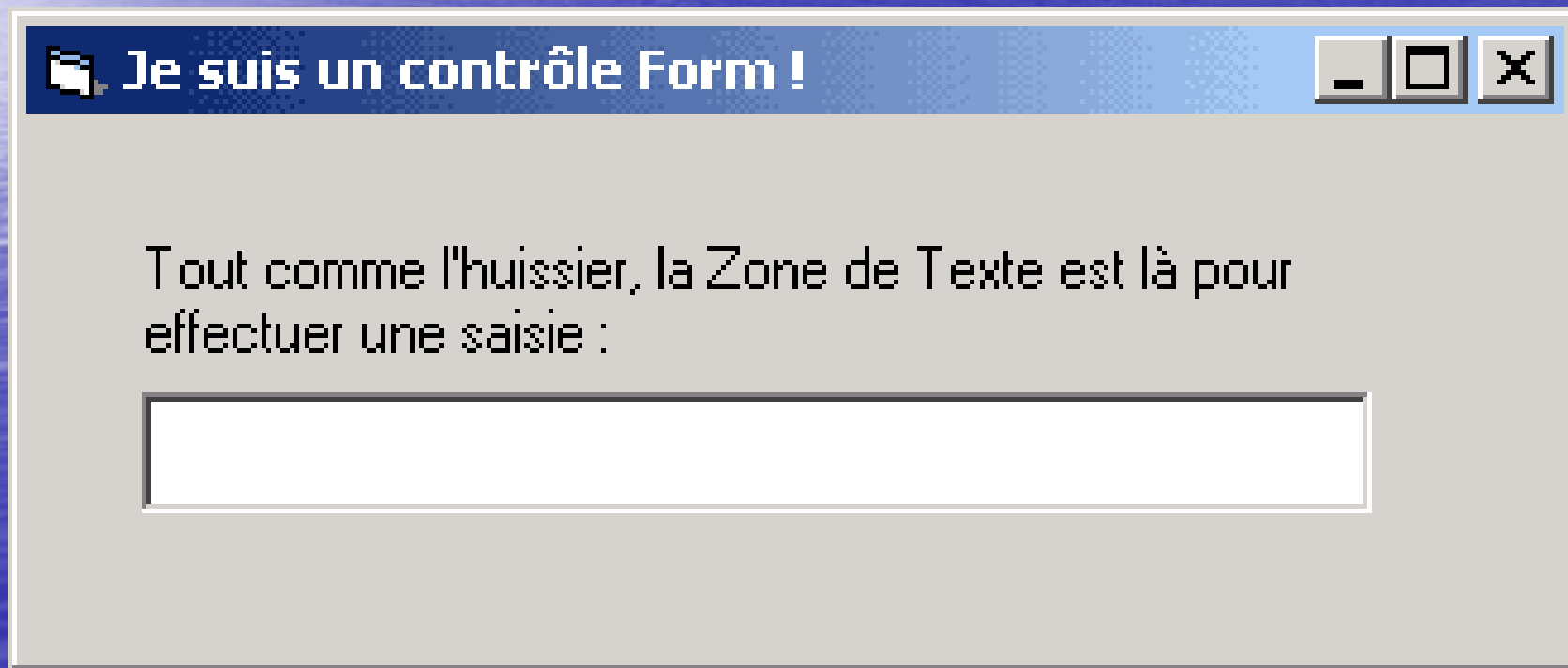


Le Label (Etiquette)

- Ce qu'il faut comprendre avec les Labels, c'est qu'ils ne peuvent jamais servir à effectuer une saisie par l'utilisateur.
- Pour le reste, les propriétés notables d'un contrôle Label sont :
- Name : bien sûr
- Caption : évidemment
- Alignment : qui règle l'alignement du texte (gauche, centré, droite)
- BorderStyle : qui désigne le type de bordure
- Quant aux événements possibles sur un contrôle Label, disons en première approche qu'ils incluent le Click déjà vu, plus d'autres événements dont nous ne parlerons que plus loin.

La Zone de Texte (TextBox)

- Ces zones (de la classe "TextBox" pour VB) peuvent servir à saisir une information. Il s'agit du seul contrôle permettant une saisie au clavier par l'utilisateur. En Visual Basic, il n'y a donc plus à proprement parler d'instruction Lire. A la place de cette instruction, on est contraint de passer par de telles zones.



Je suis un contrôle Form !

Tout comme l'huissier, la Zone de Texte est là pour effectuer une saisie :

La Zone de Texte (TextBox)

- La seule chose vraiment importante à savoir est que toute information contenue dans une zone de texte est obligatoirement de type... texte ! (autrement dit, cela inclut le cas où il s'agit d'un nombre). Conclusion, l'emploi de fonctions de conversion s'avèrera fréquemment indispensable.
- **Propriétés :**
- La propriété essentielle d'une Zone de Texte est... Text. C'est la propriété qui désigne son contenu. Comme toute propriété, elle va pouvoir être utilisée tant en lecture qu'en écriture.
- Supposons ainsi que nous avons défini une zone de classe TextBox, que nous avons appelée "NomDeFamille". Pour mettre cette zone à blanc à l'affichage de la feuille de dialogue, on écrira tout simplement :
- `Nomdefamille.Text = ""`
- Une fois que l'utilisateur aura entré quelque chose dans cette zone, si l'on veut récupérer ce quelque chose dans la variable Toto, on passera l'instruction suivante :
- `Toto = Nomdefamille.Text`

La Zone de Texte (TextBox)

- Autres propriétés intéressantes des zones de texte :
- Multiline : autorise ou non l'écriture sur plusieurs lignes
- Scrollbars : fait figurer dans la TextBox une barre de défilement horizontale ou verticale (ou les deux)
- PasswordChar : crypte le texte entré par le caractère stipulé (généralement, on choisit le caractère *)
- MaxLength : limite le nombre de caractères qu'il est possible de saisir dans la zone de texte.
- On retrouve bien entendu à propos des zones de texte les événements déjà aperçus à propos des contrôles précédents, comme le Click. Toutefois, un nouvel événement fait son apparition : il s'agit de Change, qui survient chaque fois que le contenu de la zone de texte est modifié. Cet événement peut donc survenir :

La Zone de Texte (TextBox)

- soit parce que l'utilisateur a frappé un texte dans la zone - et dans ce cas, il provoque un événement Change à chaque nouveau caractère frappé
- soit parce qu'une ligne de code provoque une modification de la propriété Text de cette zone.
- Il faut donc bien réfléchir avant d'écrire une procédure liée à une zone de texte : quand veut-on qu'elle se déclenche ? Et même, souhaite-t-on réellement déclencher une procédure à chaque fois que quelque chose est modifié dans cette zone ?

Partie 3

Quelques Fonctions

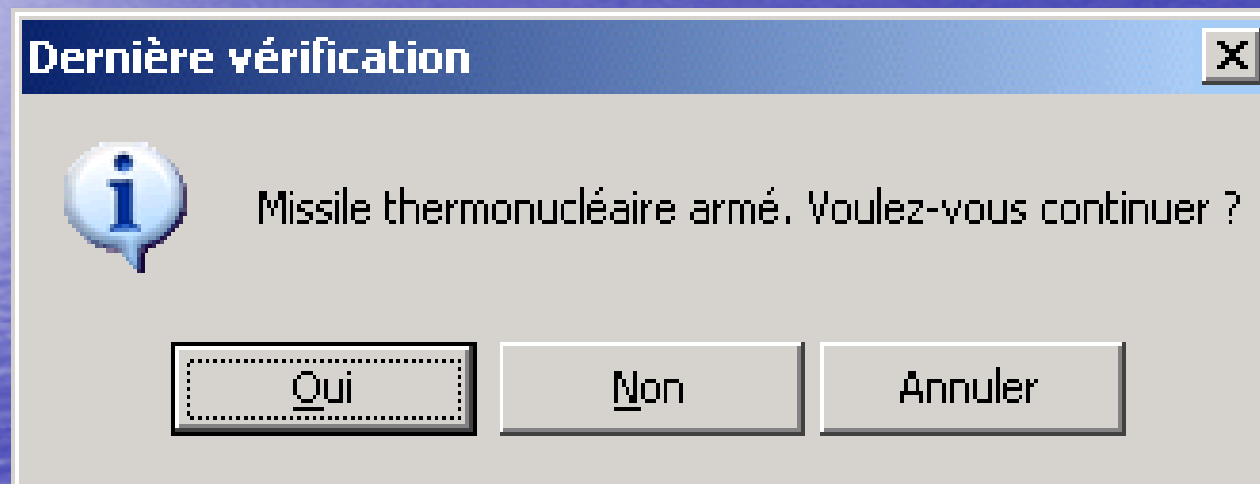
- **Les Grands Classiques**
- On va bien entendu retrouver dans VB les fonctions standard présentes dans quasiment tous les langages. Je passe sur les fonctions mathématiques (trigonométriques ou autres...) pour inventorier rapidement les fonctions plus "algorithmiques":
- **Traitement des chaînes**
- Mid (Nomdechaîne, nombre1, nombre2) : renvoie une chaîne, extraite de Nomdechaîne, commençant au caractère numéro nombre1 et faisant nombre2 caractères de long
- Len (Nomdechaîne) : renvoie le nombre de caractères de Nomdechaîne.
- Et bien d'autres ! Quelques nouveautés à ce propos :

- LTrim (Nomdechaîne) : renvoie la chaîne Nomdechaîne, débarrassée de tous les espaces se trouvant à gauche.
- Rtrim (Nomdechaîne) : renvoie la chaîne Nomdechaîne, débarrassée de tous les espaces se trouvant à droite.
- AllTrim (Nomdechaîne) : renvoie la chaîne Nomdechaîne, débarrassée de tous les espaces se trouvant à droite et à gauche.
- Ces trois fonctions sont particulièrement utiles dans le traitement des fichiers texte
- **Fonctions numériques :**
- Int (nombre) : renvoie la partie entière de ce nombre
- Rnd () : renvoie un nombre pseudo-aléatoire compris entre 0 (inclus) et 1 (exclu). NB : pour que Rnd () ne renvoie pas le même nombre, ou la même série de nombres, à chaque exécution, il faut systématiquement le faire précéder de l'instruction Randomize (toute seule, sur une ligne)
- Val (Chaîne) : renvoie un nombre si Chaîne est composée de chiffres
- Str (Nombre) : renvoie Nombre sous forme de chiffres (c'est-à-dire de caractères)

Une drôle de bête : la fonction MsgBox

- Comment envoyer facilement des informations à l'utilisateur ? La réponse à cette question qui vous taraude se trouve dans les lignes qui suivent.
- Bien sûr, si vous souhaitez envoyer des messages riches, compliqués et/ou présentés d'une manière originale, la seule solution reste de programmer des Form correspondant à vos souhaits et de passer les instructions VB nécessaires pour qu'elles contiennent les informations voulues.
- Mais si votre seule ambition, à un moment donné d'une application, est d'envoyer un message à l'utilisateur qui se limite à un texte, ou à un chiffre, ou même à une combinaison des deux, avec comme seuls boutons possibles un bouton OK, Annuler, etc., alors VB met à votre disposition la fonction MsgBox.

- Vous pourrez ainsi à moindre frais envoyer à l'écran des mini-boîtes de dialogue d'un type bien connu par tous les utilisateurs de Windows, du style :



- L'originalité de la chose, qui peut quelque peu dérouter au début, est donc que l'on passe par une fonction pour exécuter quelque chose qui ressemble à une instruction Ecrire en algorithmique. Mais, en réalité, le paradoxe n'en est pas un. Car une petite boîte de dialogue comme celle présentée ci-dessus, si elle envoie bel et bien un message à l'utilisateur, récupère également une réponse de cet utilisateur (sur quel bouton a-t-il appuyé). Dès lors, on se dit que quelque chose qui comporte plusieurs paramètres en entrée et une valeur en sortie, a toutes les chances d'être considéré comme une fonction par un langage... CQFD !
- La syntaxe de la fonction MsgBox est :
- Variable = MsgBox (texte1, integer, texte2, ...)
- La fonction MsgBox comporte donc trois arguments essentiels, qui sont respectivement :
- le texte du message à envoyer
- le style des boutons et de l'icône éventuelle à faire figurer sur le message
- le texte de la barre de titre

- En ce qui concerne les deux arguments de type texte, aucun problème. En revanche, le deuxième argument, de type Integer, mérite qu'on s'y arrête. Cet Entier (Integer) va donc avoir pour rôle de spécifier tout à la fois le type d'icône employée dans la boîte de message, et l'échantillon des boutons proposés à l'utilisateur. Toutefois, cet argument peut être spécifié de deux manières différentes qu'il convient de connaître :
- Chaque possibilité d'icône ou de boutons est associée à un nombre entier (voir l'Aide du logiciel). Pour obtenir la combinaison voulue, il suffit d'additionner les nombres correspondants, et c'est cette somme qui sera le deuxième argument.
- l'autre façon de spécifier cet argument consiste à employer des Constantes VB. Ces Constantes VB sont des mots réservés du langage, qui sont traduits par le compilateur en nombre entiers. Du point de vue du résultat, cette solution est donc strictement équivalente à la précédente, où on entrait directement des nombres. En revanche, certains programmeurs la préféreront en raison de la plus grande lisibilité qu'elle introduit dans le code. A noter que Ces Constantes VB ne s'inventent pas ! Elles figurent dans l'Aide du logiciel, et c'est là qu'il faut les chercher (elles apparaissent toutefois sous forme de liste déroulante lorsqu'on entre le code dans l'éditeur. Avec un tout petit peu d'habitude, le choix en est grandement facilité...

- A titre d'illustration, la boîte de dialogue située au-dessus pourrait être indifféremment programmée par :
- Texte = "Enregistrer les modifications (...) ?"
Titre = "Microsoft FrontPage"
Toto = MsgBox (Texte, 51, Titre)
- Ou par :
- Texte = "Enregistrer les modifications (...) ?"
Titre = "Microsoft FrontPage"
Toto = MsgBox (Texte, vbExclamation + vbYesNoCancel, Titre)
- Tout cela n'épuise pas la question de la valeur renvoyée par la fonction MsgBox, valeur qui va donc se retrouver dans la variable Toto. On a vu que cette valeur correspondait au bouton sur lequel l'utilisateur aura appuyé pour clore la boîte de message.
- On pourra donc tester la valeur de la variable Toto pour déterminer quel était ce bouton (s'il y en avait plus d'un, évidemment...).
- Là aussi, ce test peut-être écrit soit sous la forme d'une valeur entière, soit sous celle d'une constante VB, les renseignements à ce sujet se trouvant dans l'Aide. En l'occurrence, on pourrait avoir à la suite du code précédent des lignes du genre :
- If Toto = 6 Then
 ' enregistrement
Elseif Toto = 7 Then
 ' pas d'enregistrement
Else
 ' annulation
EndIf

- Ou bien :
- ```
If Toto = VbYes Then
 ' enregistrement
Elseif Toto = VbNo Then
 ' pas d'enregistrement
Else
 ' annulation
EndIf
```
- En résumé, MsgBox est une fonction qui vous donne accès à un type de Form préprogrammé, aux fonctionnalités limitées, mais à la programmation très rapide.
- La seule exception à tout ce qui précède concerne l'utilisation de MsgBox avec toutes les options par défaut : pas d'icônes, juste un bouton OK, et juste un message (pas de titre). La syntaxe peut être alors considérablement simplifiée pour donner :
- ```
MsgBox Truc
```
- ...Qui affichera donc la valeur de la variable Truc dans une boîte. Si cette utilisation de MsgBox produit un résultat nettement trop grunge pour figurer dans un produit fini, elle peut parfois s'avérer pratique pour effectuer certains déboguages.

Partie 4

contrôles et groupes

- Les Cases
- Il existe sous Windows deux sortes de cases :
- les cases dites "cases à cocher" (Checkbox): Elles sont carrées, et indépendantes les unes des autres, même si elles sont regroupées dans un cadre pour faire plus joli.
- les cases dites "cases d'option", voire "boutons radio" (OptionButton). Elles sont rondes et font toujours partie d'un ensemble (dessiné par l'objet Frame). Au sein d'un ensemble de cases d'option, jamais plus d'une seule case ne peut être cochée à la fois.

Illustrations :

Cases à cocher (CheckBox)

Questionnaire de santé me...

- Je ne rate jamais Loft Story
- Je trouve Joey Starr très sympathique
- J'adore la musique, surtout la techno

OK

Cases d'option (OptionButton)

The image shows a Java Swing window titled "Sondage" with a standard title bar (minimize, maximize, close buttons). Inside the window, there is a question: "Quel est votre poète préféré ?". Below the question is a vertical list of five radio buttons, each followed by a poet's name: Guillaume Apollinaire, Jacques Prévert, Charles Baudelaire, Arthur Rimbaud, and Ernest Antoine Seillière. The "Jacques Prévert" option is selected, indicated by a filled radio button and a dashed rectangular border around the text. Below the list is a large rectangular button labeled "OK".

Cadre (Frame)

- Moralité, avant de mettre des cases et d'écrire le code qui s'y rapporte, il faut bien se demander de quel type de cases on a besoin.
- Une fois ce préambule posé, il n'y a pas de problèmes particuliers, hormis que pour des cases d'option, il faut toujours créer le Frame avant de poser des cases à l'intérieur de ce Frame. Dans le cas contraire, le langage ne reconnaîtra pas les cases comme faisant partie du même ensemble, et elles auront un comportement erratique.
- Sinon, la propriété la plus intéressante d'une case est de loin celle qui nous permet de savoir si elle est cochée ou non. Cette propriété s'appelle Value.
- ◦ Value prend la valeur True ou False lorsqu'elle concerne les cases d'option.
- ◦ Value prend la valeur 1 ou 0 lorsqu'elle s'applique aux cases à cocher. Cette différence de traitement peut paraître aberrante, mais il faut se rappeler que les cases à cocher peuvent aussi posséder un état intermédiaire grisé. Auquel cas leur propriété Value prend la valeur 2.
- Comme toute propriété, Value peut être utilisée en écriture (pour initialiser telle ou telle case au moment où elle arrive à l'écran) ou en lecture (pour tester quelle case a coché l'utilisateur, et accomplir du coup un traitement ou un autre).

Les groupes de contrôles

- Avec les deux exercices qui précèdent, on constate un alourdissement considérable du code, qui devient redondant pour un certain nombre d'objets.
- Cela tient au fait que jusqu'à présent, nous avons toujours considéré que chaque contrôle était un objet complètement indépendant de ses congénères. Cela se manifestait de deux manières :
- lors de leur création, nous allions toujours chercher l'objet dans la boîte à outils, en nous abstenant soigneusement de procéder à des copier coller.
- par conséquent, chaque objet possédait une propriété Name bien à lui, qui nous permettait de le désigner sans ambiguïté en écrivant le code. Et du coup, chaque objet possédait ses propres procédures événementielles, distinctes de celles liées aux autres objets.

- Mais cette stratégie induit lors de certains programmes une lourdeur qui ne vous aura pas échappé. Lorsque plusieurs objets de même type remplissent des tâches voisines, voire semblables (cf. les deux exercices précédents), on se retrouve avec une flopée de procédures événementielles (une par objet, parfois davantage) qui se ressemblent beaucoup, voire qui sont carrément semblables.
- La création d'un groupe de contrôles a pour but d'alléger tout ceci. Comment ça marche, me direz-vous ? Eh bien voici.
- Lors de la création des contrôles (mettons, une série de quatre cases à cocher), on procède dorénavant par copier-coller, en prenant soin, s'il s'agit de cases d'options, que le collage s'effectue bien au sein du Frame (sinon, elles ne marcheront pas correctement ensemble). A la question "souhaitez-vous créer un groupe de contrôles?", on répond sans réserves par l'affirmative.
- Dès lors, les quatre objets ainsi créés seront en fait des membres d'un même groupe. En Français, cela signifie que ces quatre objets porteront le même nom (Name) mais affublé d'un Index qui servira à les identifier individuellement. Tout cela rappelle furieusement les tableaux, hein ? C'est bien normal, puisque c'en est.
- Si j'osais une métaphore ailée (mais non filée), je dirais que le groupe est au contrôle ce que le tableau est à la variable simple. Limpide, non ? J'ajoute pour conclure que Index est une propriété – c'est logique – de tout élément d'un groupe de contrôles, définissant son indice (propriété à ne surtout pas confondre avec TabIndex, qui définit son ordre dans la tabulation, ce qui n'a rien à voir avec la choucroute).

- Donc, résumons nous.
- Si on a créé un groupe de 4 contrôles de classe Checkbox le nom par défaut de ce groupe sera Check1. Rebaptisons-le CaseCarrée pour plus de clarté. On aura alors CaseCarrée(0), car l'Index commence à zéro comme pour tout tableau qui se respecte, CaseCarrée(1), CaseCarrée(2) et CaseCarrée(3).
- Si on a un traitement à effectuer sur les quatre cases, on pourra dorénavant écrire une boucle, par exemple pour décocher tout ça :
- For i = 0 to 3
CaseCarrée(i).Value = 0
Next I
- Déjà, on se marre bien. Mais c'est encore plus rigolo quand on crée la procédure associée à ce groupe. Cela produit un intitulé du genre :
- Private Sub CaseCarrée_Click (Index as Integer)

End Sub

- Eh oui, vous avez bien vu. Cela signifie que dorénavant, pour un événement donné, il n'y a plus qu'une seule procédure pour l'ensemble des contrôles du groupe !
- Quant à la variable Index, dont vous voyez que Visual Basic la crée automatiquement en tant que paramètre en entrée de cette procédure, elle contient à chaque exécution de la procédure, l'indice de l'élément du groupe qui vient de déclencher l'événement.
- Ainsi par exemple, si au sein de cette procédure, on a besoin de savoir quelle case a été cliquée, il suffit de tester la variable Index, qui, en l'occurrence, vous l'aurez deviné, peut valoir de zéro à trois.
- A noter qu'Index est ici une simple variable, à ne pas confondre avec la propriété Index des éléments des groupes de contrôles. Donc, si ce nom de variable vous semble trop lourdingue, rien ne vous interdit de le modifier.
- Formulons tout cela d'une traite : Index est le nom par défaut du paramètre en entrée pour la procédure événementielle, paramètre qui prend à chaque exécution la valeur de la propriété Index du membre du groupe qui a déclenché cette exécution.
- Avec tout cela, certains programmes vont pouvoir s'alléger considérablement. Moralité, en VB, c'est comme pour tout le reste, pour pouvoir être fainéants, devenons compétents.

Les Listes

deux contrôles de listes

- Les listes classiques dans Windows peuvent posséder ou non deux caractéristiques.
- Elles peuvent être modifiables : c'est-à-dire que l'utilisateur a la possibilité d'entrer un élément qui ne figure pas au départ dans la liste. Cette caractéristique concerne donc les données de la liste proprement dites.
- Elles peuvent être déroulantes : c'est-à-dire qu'on ne voit qu'un seul élément de la liste à la fois, et qu'il faut cliquer sur la flèche du côté pour "déplier" la liste. Cette caractéristique joue donc uniquement sur l'aspect de la liste, et aucunement sur la manière dont les données sont gérées.
- Une liste peut donc prendre quatre têtes, selon qu'elle est modifiable ou non, déroulante ou non ("chez les papous, y a les papous à poux et les papous pas à poux, etc.")
- VB fournit deux contrôles de liste :

- une liste dite simple : contrôle ListBox
- une liste dite modifiable : contrôle ComboBox
- **Propriétés indispensables :**
- ListIndex : renvoie ou définit l'indice de l'élément actuellement sélectionné. En fait, en interne, VB gère les listes un peu à la manière des tableaux. Il attribue donc à chaque élément d'une liste un indice, cet indice commençant toujours à zéro. A noter que si aucun élément n'est sélectionné dans la liste, la propriété Listindex vaut -1.
- List : Savoir quel est l'indice de l'élément sélectionné dans une liste, c'est bien. Mais savoir à quel texte correspond cet indice, c'est mieux ! Cette propriété renvoie en clair (sous forme de texte) un élément d'une liste en fonction de son indice. Il s'agit donc, chose assez rare, d'une propriété qui nécessite un argument (le numéro d'indice).

- Pour récupérer sous forme de texte l'élément actuellement sélectionné dans la liste appelée Produits, on écrira donc le code suivant :
- NuméroProduit = Produits.ListIndex
NomProduit = Produits.List(NuméroProduit)
MsgBox "Vous avez choisi le produit : " & NomProduit
- Ou plus directement :
- NomProduit = Produits.List(Produits.ListIndex)
MsgBox "Vous avez choisi le produit : " & NomProduit
- Toutefois, il faut noter que les contrôles ComboBox nous simplifient grandement la vie, puisqu'à la différence des ListBox, ils nous proposent directement une propriété Text. Ainsi, si Produits est un contrôle ComboBox, et uniquement dans ce cas, on pourra écrire :
- NomProduit = Produits.Text
MsgBox "Vous avez choisi le produit : " & NomProduit

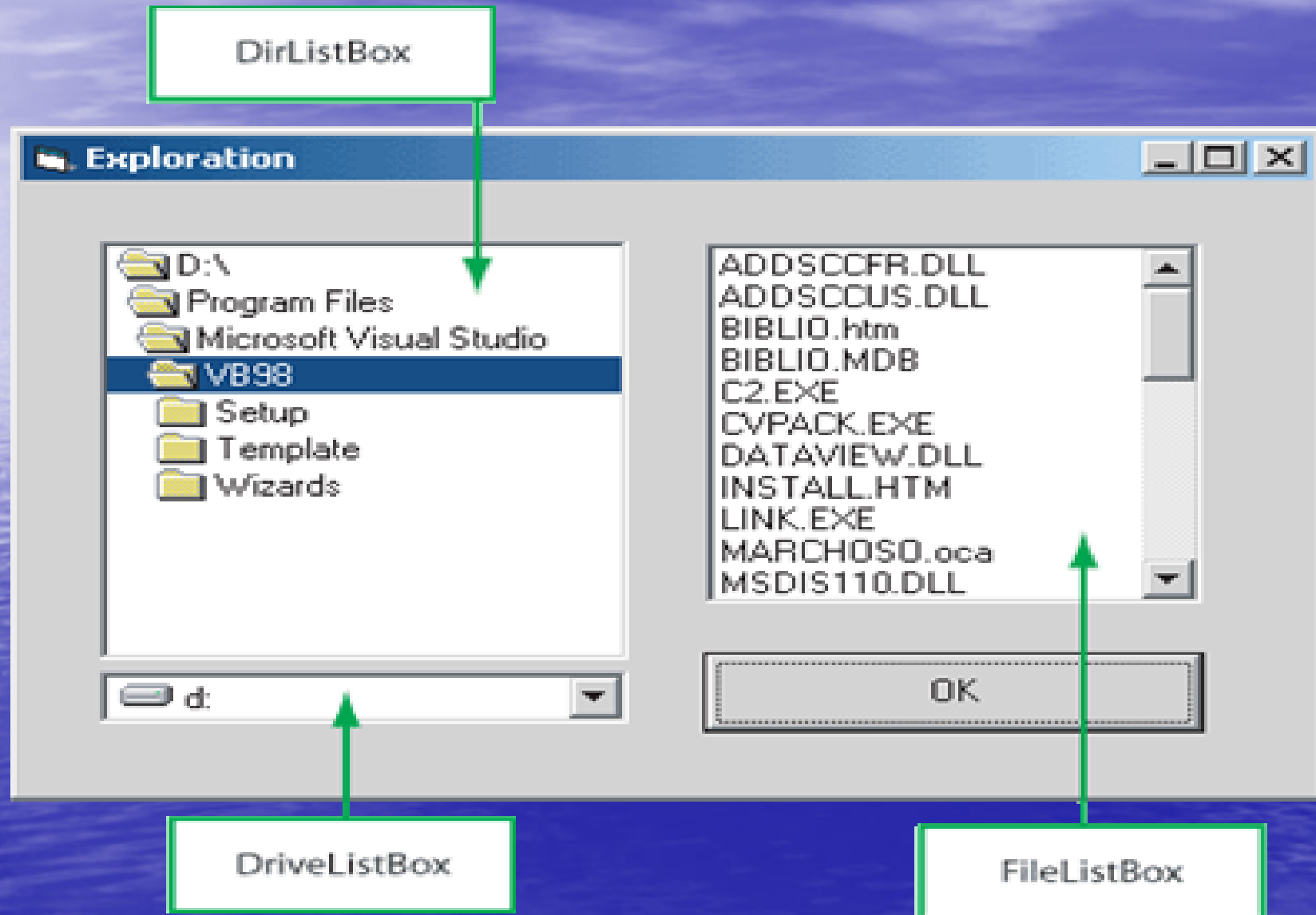
Autres propriétés intéressantes :

- ListCount, qui renvoie le nombre d'éléments d'une liste (propriété numérique)
- Multiselect, qui permet la sélection multiple (propriété booléenne)
- Sorted, qui trie automatiquement les éléments d'une liste (propriété booléenne)
- **Méthodes à connaître**
- Nous découvrons avec les contrôles ListBox et ComboBox des méthodes qu'il est indispensable de connaître afin de gérer des objets de type liste :
- AddItem Chaîne : ajoute l'élément Chaîne à une liste (un argument supplémentaire, facultatif, permet éventuellement de spécifier à quel indice l'élément doit être inséré).
- RemoveItem (indice) : supprime de la liste l'élément possédant l'indice spécifié.
- Clear : efface tous les éléments d'une liste

Trois autres contrôles pour des usages spéciaux

- Il n'est pas rare que dans une application, on doive permettre à l'utilisateur de naviguer dans le disque dur, les répertoires, les fichiers, etc. (comme dans la commande Fichier – Ouvrir de tout logiciel).
- Pour réaliser cela, on va avoir recours à trois contrôles supplémentaires, spécialisés, dont le rôle est de reproduire des outils familiers sous Windows :
- DriveListBox : qui fournit automatiquement la liste des lecteurs logiques disponibles sur la machine où tourne l'application.
- DirListBox qui fournit automatiquement la liste des répertoires contenus dans une unité donnée
- FileListBox qui fournit automatiquement la liste des fichiers contenus dans un répertoire donné.

Illustration :



- La présence de ces contrôles soulage votre tâche, mais ne vous épargne toutefois pas totalement d'écrire un chouïa de code, afin de les faire fonctionner ensemble. Ces trois objets étant des cas particuliers de contrôle liste, s'y appliquent éventuellement List, Listcount et Listindex.
- Mais pour gérer la mise à jour de ces contrôles les uns en fonction des autres, les propriétés suivantes sont indispensables (et suffisantes) :
- Drive : qui renvoie ou définit l'unité logique actuellement en vigueur dans un contrôle DriveListBox
- Path : qui renvoie ou définit le chemin actuellement en vigueur dans un contrôle DirListBox ou dans un contrôle FileListBox
- Filename : qui renvoie ou définit le fichier actuellement sélectionné dans un contrôle FileListBox.

Partie 5

Éléments Graphiques

- VB étant un langage objet, donc événementiel, il exploite l'interface Windows. Alors, autant mettre de la couleur, des images, et tout ce qui rend la vie plus agréable. Surtout que ce n'est franchement pas difficile.
- 1. Utiliser (bêtement) la méthode Print
- `Form1.Print "Coucou"`
- Ira écrire "Coucou" en haut à gauche de la Form. La prochaine instruction Print provoquera une écriture juste en dessous... sauf si nous avons terminé l'instruction précédente par un point-virgule. Auquel cas, la prochaine écriture s'effectuera à la suite.
- **Remarque** : A priori, si l'utilisateur redimensionne la Form, le texte précédemment inscrit disparaîtra, ce qui peut être gênant. Il est facile d'y remédier : fixez auparavant la propriété `AutoRedraw` de la Form à `True`. Attention toutefois, cela ralentit l'application.
- **Autre Remarque** : `Print` s'applique également aux contrôles `PictureBox`. Cela représente une alternative par rapport aux modifications de `Label.Caption` que nous avons utilisé jusque là.
- **Dernière Remarque** : utiliser la méthode `Print` revient de fait à utiliser VB comme un bon vieux langage non graphique. Autrement dit, je ne l'ai signalée que pour le principe, il y a bien mieux à faire que se servir de ce bidule.

Spécifier les couleurs

- Dans les exercices qui ont jalonné ce cours, nous avons croisé, sans nous y arrêter, un certain nombre de propriétés gérant la couleur des différents contrôles : ForeColor, BackColor, etc.
- Ce qu'il faut savoir, lorsqu'on passe une instruction pour affecter une valeur à l'une de ces propriétés, c'est qu'il existe pas moins de trois systèmes de couleurs en Visual Basic, systèmes que vous pouvez utiliser indifféremment.
- les huit couleurs de base, gérées par VB lui-même. Vous en trouverez la liste complète dans l'aide, ainsi que les Constantes VB qui désignent ces couleurs. Cela donnera des instructions du genre :
- `Form1.BackColor = VbBlack`
- Les seize couleurs de la palette, accessibles par la fonction `QBColor(i)`, *i* désignant bien entendu le code de la couleur voulue. Là aussi, il suffit d'aller voir l'aide pour obtenir la liste des couleurs disponibles. Par exemple, pour mettre la Form en vert clair (c'est immonde, mais tant pis) :
- `Form1.BackColor = QbColor(10)`

- les 16 millions de couleurs du web. Là, attention, il y a deux manières de procéder. La première est celle qu'emploie spontanément VB, et elle est calquée sur le principe du code HTML : trois fois deux caractères, représentant respectivement le codage de 256 niveaux de rouge, de vert et de bleu, en hexadécimal (si vous n'avez rien compris à cette phrase, lisez celle qui suit. Si vous l'avez comprise, lisez quand même celle qui suit).
- Le problème est que VB s'amuse à bidouiller tout seul ce codage à six caractères, exigeant qu'il soit précédé par les signes "&H" et s'amusant à l'occasion à faire sauter certains des six caractères susnommés. Alors, un bon conseil, en Visual Basic, évitez comme la peste le codage HTML et utilisez à la place la fonction RGB. Celle-ci fonctionne exactement sur le même principe, mais les doses de rouge, de vert et de bleu, fournies en argument, sont spécifiées sous forme d'entiers. Ainsi, pour repeindre la Form en rouge vif, on écrira :
- `Form1.BackColor = RGB(255, 0, 0)`

- Les coordonnées
- Dès que l'on va placer ou déplacer des objets graphiques, il va falloir spécifier les coordonnées. Et ce n'est pas toujours d'une simplicité biblique.
- Premier point, sur lequel M. de Lapalisse, un ancien voisin à moi, ne m'aurait pas démenti : l'écran possède deux dimensions. Cette forte pensée étant posée, il reste que l'origine (le point 0, 0) se trouve en haut à gauche, et pas du tout en bas à gauche, là où tout occidental, hormis Bill Gates, l'aurait placé.
- Deuxième problème : comment savoir quelles sont les dimensions de l'objet sur lequel on veut dessiner ? Là, il faut faire attention. Car si Height et Width sont, comme vous le savez, les dimensions extérieures d'un objet, dans le cas d'une Form, on peut avoir besoin de connaître les dimensions intérieures. Celles-ci nous sont alors fournies par les propriétés ScaleHeight et ScaleWidth de cette Form.
- Autre aspect : on dispose de plusieurs unités de mesures possibles pour les coordonnées.

- L'unité par défaut est le twip. Cette chose représente très exactement 1/567e de centimètre. Le gros avantage de cette unité, c'est qu'elle est indépendante de la résolution du périphérique utilisé. Un centimètre, c'est toujours un centimètre (nous négligerons ici les effets de la relativité, la vitesse moyenne de déplacement des écrans par rapport à celui qui les regarde étant généralement négligeable par rapport à la vitesse de la lumière).
- Mais il y a aussi le pixel. L'avantage du pixel, c'est sa rapidité. Vous épargnez à Windows une conversion, puisque un pixel, c'est un point. Mais l'inconvénient, c'est que selon les résolutions d'écran, le nombre de pixels varie fortement... (le nombre de twips varie aussi selon la taille de l'écran, me direz-vous, alors perdu pour perdu...)
- On modifie (ce qui n'est généralement pas indispensable) le système de mesures par la propriété ScaleMode de la Form.

Contrôles graphiques et images

- Certains contrôles sont dédiés à l'affichage de graphiques, d'une manière ou d'une autre. Il est maintenant temps de s'y intéresser de plus près...
- 4.1 contrôles graphiques
- On doit mentionner pour mémoire, mais vraiment uniquement pour cela :
- le contrôle Shape, qui peut prendre des formes géométriques simples : lignes, rectangles, cercles, ellipses.
- le contrôle Line, qui se contente de figurer une ligne.
- L'utilisation de ces deux contrôles demeure cependant marginale. S'ils peuvent éventuellement servir à faire de petits schémas, ils sont incapables de produire de vrais effets graphiques. Mais surtout, ces contrôles sont très peu employés parce qu'ils ne peuvent recevoir aucun événement souris. Leur intérêt pour un programme est donc fatalement extrêmement limité.
- 4.2 contrôles image
- La première chose à dire si l'on a besoin d'une image, c'est qu'on peut la mettre directement en fond d'une Form, via la propriété Picture. Les boutons de commande, eux aussi, possèdent

- cette propriété Picture, et peuvent donc servir de support à une image - à condition, je le rappelle, que leur propriété Style ait auparavant été réglée sur Graphical.
- Toutefois, on peut avoir besoin d'images qui ne s'identifient pas avec une Form ou un Bouton. Dans ce cas, des contrôles adéquats sont là pour nous sauver la vie.
- Ce sont des contrôles de VB, exactement comme les zones de textes, les boutons, etc., mais qui ont pour but de contenir des jolies images. Enfin, des images. Il y en a deux : le contrôle PictureBox et le contrôle Image.

- De ces trois critères, sauf cas très spécial, seuls les deux premiers sont réellement importants. Conclusion, soit on a juste besoin d'une image, et alors il faut choisir un contrôle Image. Soit on a besoin d'un conteneur incorporant d'autres choses en plus de l'image, et alors il faut choisir le PictureBox.
- La propriété essentielle d'un contrôle Image est Picture, qui permet de désigner le fichier graphique qui viendra s'incruster dans le contrôle.
- 4.3 Méthodes graphiques
- Il existe un certain nombre d'actions dites méthodes graphiques, qui ne sont pas des contrôles, mais qui sont du code qui a pour rôle de tracer des dessins sur un objet (une Form, notamment). Le dessin s'effectue donc au cours de l'exécution du programme. Donc, cela gagne de l'occupation mémoire par rapport à un contrôle similaire.
- Ce n'est pas que cela serve beaucoup, mais par acquit de conscience, allons-y :

- Pset : qui permet de tracer des pixels un par un
- Line : qui permet de tracer des lignes
- Circle : qui dessine cercles, arcs de cercles et ellipses
- Pour la syntaxe précise, je vous renvoie à l'aide, bien que je doute que tout cela vous serve fréquemment.

Partie 6

- **Objets, propriétés, événements**
- Ce chapitre fourre-tout a pour but de faire un tour relativement complet des propriétés et des événements les plus utiles pour les différents objets. Bien entendu, vous avez déjà eu l'occasion de croiser un certain nombre de ces choses, mais là, vous allez repartir avec de quoi faire votre petit marché en toute quiétude.

Deux objets de curiosité

- Il s'agit du petit chronomètre situé dans la barre de contrôles. Ce contrôle joue un rôle fondamental et indispensable : c'est lui qui va permettre d'effectuer des traitements, à intervalles fixés d'avance, indépendamment des actions effectuées par l'utilisateur.
- Jusqu'à présent, en effet, le code ne s'exécutait (si l'on met de côté le `Form_Load` et une éventuelle procédure principale située dans un module) qu'au cas où l'utilisateur faisait quelque chose : saisie au clavier, mouvement ou clic de souris, etc. Or, on peut tout à fait avoir besoin, dans certaines applications, que le code se mette en route, même si l'utilisateur ne fait rien du tout.
- Le contrôle `Timer`, graphiquement toujours invisible, va générer des événements, des "tops", à une cadence choisie par le programmeur, qui déclencheront la procédure `NomDuTimer_Timer()`. Le programmeur pourra donc mettre dans cette procédure tout ce qui doit se passer indépendamment de ce que fera – ou ne fera pas – l'utilisateur.

- Un Timer est très utile dans certains jeux, soit qu'on ait besoin de chronométrer quelque chose, soit qu'on ait besoin de provoquer certains événements à intervalles réguliers (un déplacement d'un bidule, la survenue d'un événement du jeu tiré au hasard, etc.)
- La propriété essentielle d'un Timer est Interval. C'est elle qui fixe l'intervalle entre chaque top, exprimé en millisecondes).

L'objet "Application" : App

- Ce jour est à marquer d'une pierre blanche, car voici notre première rencontre avec un objet qui n'est pas un contrôle, et qui ne possède donc aucune existence sous forme de représentation graphique. Loin des yeux, mais pas loin du cœur, voici l'objet App, qui n'est autre que l'ensemble de votre application elle-même, en personne. App, comme tout objet, possède un certain nombre de propriétés. L'une d'entre elles est particulièrement utile : il s'agit de la propriété Path, accessible uniquement en lecture, et qui indique quel est le chemin d'accès de l'exécutable.
- Ceci se révèle particulièrement utile quand on a besoin d'aller chercher d'autres documents (fichiers textes, images à charger, etc.) dans une application. On ne sait jamais a priori comment s'appelle le répertoire dans lequel votre application a été installée sur d'autres machines que la vôtre. Grâce à l'objet App, et à sa propriété Path, on va donc pouvoir récupérer l'emplacement de l'exécutable, et de là, pointer le chemin conduisant au fichier que l'on veut manipuler.
- Donc, en résumé, on peut utiliser cette propriété pour désigner un fichier par référence relative au répertoire dans lequel a été installé l'application, quel que soit le répertoire en question.

- Admettons par exemple que vous ayez besoin d'aller désigner un fichier au cours d'un programme, parmi une dizaine disponibles (genre mois01.txt, mois02.txt, etc.). Vous mettez donc le nom du fichier à aller chercher dans une variable nommée Fic. Mais comment préciser à l'ordinateur le chemin menant à ce fichier ? Ce chemin sera a priori différent selon la manière dont votre application aura été installée sur chaque machine. La parade consistera donc :
- à obliger l'application à installer la série des fichiers txt toujours au même endroit par rapport à l'exécutable (on supposera ici qu'ils sont dans un sous-répertoire nommé Files). On verra plus loin comment réaliser cette prouesse.
- à désigner dans le code le chemin du fichier en récupérant celui de l'exécutable, et en pointant ensuite le sous répertoire Files.
- Illustration :
- ' La variable Fic stocke le nom du fichier à ouvrir
Chemin = App.Path
If Right(Chemin, 1) <> "\" Then
 Chemin = Chemin & "\"
EndIf
Complet = Chemin & Fic
MsgBox "L'emplacement complet est : " & Complet

Propriétés

- On n'examinera ici que certaines des propriétés les plus courantes, celles que l'on retrouve pour la majorité, sinon la totalité, des contrôles. Bien entendu, vous avez déjà expérimenté un certain nombre de ces choses, et il ne s'agit là que d'un petit complément..
- 2.1 Localisation des contrôles
- Tout contrôle placé sur une Form possède deux propriétés qui régissent son emplacement :
- Top : distance séparant le bord supérieur du contrôle du haut de la Form
- Left : distance séparant le bord gauche du contrôle de la gauche de la Form
- Comme vous le savez aussi, il y a également deux propriétés qui règlent la taille du contrôle. Ce sont :
- Width qui spécifie sa largeur
- Height qui stipule sa hauteur

- Avec ces quatre propriétés, on règle donc et la taille et l'emplacement de tout contrôle, et on peut faire des choses graphiquement très jolies. CQFD.
- 2.2 Activation des contrôles
- La plupart des contrôles possèdent les propriétés suivantes :
- Enabled : permet / interdit l'action de l'utilisateur sur le contrôle considéré. Bon nombre de contrôles (mais pas tous) deviennent grisés lorsque cette propriété est False. Vous l'aurez deviné, il s'agit d'une propriété booléenne.
- Visible : affiche / cache le contrôle considéré (et du même coup, permet / interdit) au contrôle de recevoir des événements clavier ou souris. C'est fou ce qu'on peut faire avec ce petit machin là. Comme la précédente, cette propriété est booléenne.
- Tabindex : règle l'ordre de tabulation des contrôles dans la Form (propriété de type Integer).

- Evénements
- On va dans cette partie aborder des événements jusque là injustement dédaignés.
- Je passe rapidement sur le Click et le DbClick, qui n'ont plus de secrets pour vous. Toutefois, c'est le moment de préciser un détail important : un contrôle ne peut pas à la fois posséder une procédure liée au click (simple) et au double click, car un des deux événements intercepte toujours l'autre. clic ou double clic, le concepteur d'interface doit donc choisir entre les deux !
- Plus intéressants, car moins connus, en voici quelques autres :

- 3.1 Événements Focus

- Le focus est le fait, pour un contrôle, d'être celui qui est actuellement désigné par la tabulation (un petit liseré en pointillé apparaît autour d'un objet lorsqu'il a le focus, ou dans le cas des zones de texte, le curseur clignote à l'intérieur). Deux événements sont liés à la réception ou à la perte du focus :

- Gotfocus : quand l'objet reçoit le focus

- Lostfocus : quand l'objet perd le focus

- J'en profite négligemment au passage pour signaler qu'une instruction permet de forcer le passage du focus à tel ou tel contrôle. Il s'agit de la méthode Setfocus. Ainsi, la ligne de code :

- TrucMuche.Setfocus

- envoie de gré ou de force le focus sur le contrôle TrucMuche.

- **Événements clavier**

- On peut être amené, dans une application, à vouloir "guetter" la frappe de touches du clavier par l'utilisateur. Ceci recouvre grosso modo deux cas :
- On veut gérer la saisie caractère par caractère dans un contrôle TextBox ou ComboBox. Ceci pour contrôler au fur et à mesure la validité des caractères frappés (éliminer les chiffres, par exemple...) ou convertir les caractères au fur et à mesure les caractères frappés (les convertir automatiquement en minuscules ou en majuscules, par exemple...). Dans ce cas, on pourra gérer les événements KeyPress KeyUp ou Keydown (voir plus loin pour le choix) au contrôle concerné.
- On souhaite affecter, indépendamment de tout problème de saisie dans un contrôle, certaines actions à certaines touches du clavier. Typiquement, affecter les touches de fonctions : F1 pour ouvrir l'aide, etc. Ici, c'est à la Form que l'événement clavier devra être associé. Mais cela ne fonctionnera que si la propriété KeyPreview de la Form possède la valeur True. Dans ce cas, la Form reçoit ces événements avant les contrôles qu'elle contient.

- La différence fondamentale entre KeyPress d'une part, et Keydown et Keyup d'autre part, c'est que KeyPress considère quel caractère a été frappé (le résultat logiciel de la frappe), alors que Keydown et Keyup considèrent l'état physique du clavier.
- Par conséquent : KeyPress ne reconnaît pas les frappes ne produisant pas directement un caractère, comme les touches de fonction, les touches de modification, les touches de navigation et toutes les combinaisons de ces touches avec les modificateurs du clavier. En revanche, Keydown et Keyup gèrent parfaitement tout cela.
- Autre différence, KeyPress interprète la majuscule et la minuscule de chaque caractère comme des codes de touche distincts et, par conséquent, comme deux caractères différents. KeyDown et KeyUp, quant à eux, interprètent la majuscule et la minuscule de chaque caractère au moyen de deux arguments : keycode, qui indique la touche physique (A et a étant alors équivalents) et shift, qui indique l'état de shift + key et qui renvoie en conséquence soit A, soit a.
- Dernière chose sur ce chapitre, Keydown détecte le moment où une touche est enfoncée, KeyUp celui où elle est relâchée.
- La gestion fine des événements clavier restant une chose relativement pénible en Visual Basic, nous passons rapidement à quelque chose de beaucoup plus répandu...

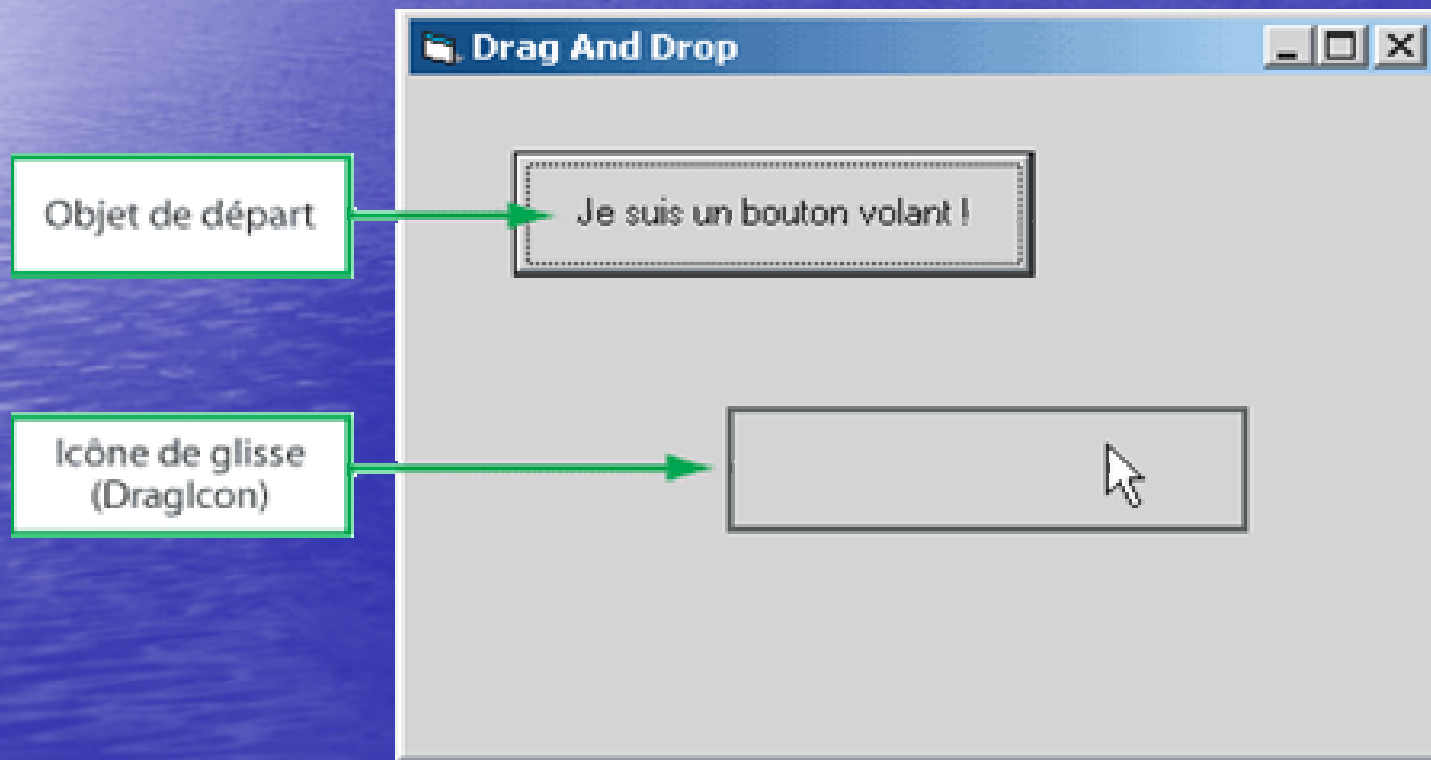
Événements souris

- MouseDown et MouseUp sont deux événements détectant respectivement le fait qu'un bouton de la souris a été enfoncé ou relâché au dessus d'un objet.
- Un aspect fondamental de ces événements, c'est que la procédure qui leur est associée possède un certain nombre de paramètres. C'était déjà le cas avec les événements claviers ci-dessus, mais j'avais alors jeté un voile pudique sur cet aspect. En tout cas, là, avec la souris, on n'y coupera pas. Les paramètres en entrée de toute procédure liée à un événement souris sont là pour vous permettre notamment de récupérer dans cette procédure :
 - quel bouton de la souris a provoqué l'événement (bouton de gauche, bouton de droite, etc.)
 - l'état des touches Maj, Ctrl et Alt du clavier. Ceci permet de gérer des combinaisons alambiquées clavier + souris. A réserver uniquement aux émules des interfaces Adobe.

- les coordonnées x et y désignant l'endroit où l'événement s'est produit. Et là, gaffe de chez gaffe ! Ces coordonnées sont toujours relatives à l'objet ayant reçu l'événement. Donc ce ne sont pas les coordonnées par rapport à la Form que si l'objet recevant l'action est bien la Form. Réciproquement, si on gère un MouseDown sur une image, le X et le Y désigneront la position de la souris par rapport à cette image au moment du MouseDown. Si l'on veut en déduire les coordonnées de la souris par rapport à la Form, il faudra ajouter les coordonnées de l'image dans le Form. Tout le monde suit ?
- Quant à l'événement MouseMove, il détecte le déplacement de la souris au-dessus d'un contrôle. Cet événement possède les mêmes paramètres que les deux événements précédents.
- Par ailleurs, il faut signaler en passant qu'en VB, les possibilités de personnaliser l'apparence de la souris sont nombreuses et simples d'utilisation. Chaque contrôle possède ainsi deux propriétés chargées uniquement de gérer cet aspect des choses :
- MousePointer: indique quelle tête la souris doit prendre lorsqu'elle survole l'objet, tête, choisie parmi la liste des possibilités standard de Windows.
- MouseIcon désigne un fichier image (type *.ico ou *.cur) définissant un pointeur de souris personnalisé.

- On peut donc très facilement, en affectant l'une ou l'autre de ces propriétés (pas les deux en même temps, ça n'aurait aucun sens), faire en sorte que le pointeur de la souris change de tête au survol de tel ou tel contrôle. C'est un des petits détails qui donnent à une interface un look professionnel et "fini", et celui-là, comme on le voit, ne coûte vraiment pas cher.
- L'enfer du jeu est un exercice un peu plus long, qui a valeur de récapitulatif. Il comporte deux difficultés majeures. La première est liée à la formule - et à l'algorithme - de calcul. Cet algorithme est posé sous forme d'exercice, et fort heureusement, de corrigé, dans le fabuleux cours d'algorithmique du même auteur. La deuxième difficulté tient à l'affichage du résultat sous un format dit "de milliers", format séparant les chiffres par groupe de trois pour améliorer la lisibilité. Cet affichage n'ayant pas été prévu par le langage VB, la programmation insère des espaces là où il faut, ce qui constitue une splendide prise de tête.

- Gérer le Cliquer - Glisser
- Là, on peut vraiment s'amuser. Et par les temps qui courent, il ne faut pas rater de si belles occasions.
- Prenons tout de suite un exemple, qui serait de permettre à un utilisateur d'effectuer un cliquer-glisser sur un bouton de commande, afin de le déplacer sur la Form.
- Regardons ce qui se passe alors :



- **Une remarque fondamentale** : on doit remarquer que le cliquer-glisser ne provoque en aucun cas de lui-même un déplacement de l'objet de départ. Si l'on veut qu'un tel déplacement survienne, on doit le programmer expressément par le code adéquat.
- La gestion du cliquer glisser n'est pas difficile, à condition de suivre méthodiquement les étapes.
- **Etape 1** : il faut autoriser l'objet considéré (ici, le bouton de commande) à subir un cliquer-glisser. Par défaut, l'utilisateur ne peut pas, en effet, cliquer-glisser les contrôles qu'il voit sur une Form. Cette autorisation s'effectue par le passage de l'instruction (qui est une méthode) :
- `NomduContrôle.Drag`
- Où doit-on écrire cette instruction ? Le plus logique, est d'autoriser l'objet à être cliqué-glissé au moment où l'utilisateur commence la manoeuvre, c'est-à-dire lorsqu'il enfonce le bouton de la souris sur l'objet. Donc, traditionnellement, on placera cette ligne de code au début de la procédure `Command1_MouseDown`

- **Etape 2** : il faut définir, si on le souhaite, quelle doit être la physionomie du cruseur de la souris durant le cliquer-glisser, autrement dit quelle est "l'icône de glisse" de l'objet. Dans le cas représenté ci-dessus, on s'est contenté de conserver l'icône par défaut pour un bouton de commande. Mais on pourrait tout à fait préciser une autre icône, en affectant la propriété DragIcon de l'objet.
- `NomduContrôleGlissé.DragIcon = image`
- De même que ci-dessus, cette instruction est elle aussi habituellement placée au moment où se déclenche le Cliquer-Glisser, c'est-à-dire au début de la procédure `NomduContrôle_MouseDown`.
- Attention, le fichier image doit obligatoirement être de type icône (*.ico), à l'exclusion de tout autre ! Toutefois, plutôt que pointer un fichier contenant l'image voulue, il est toujours plus facile de stocker celle-ci à l'avance dans un contrôle image, en le rendant au besoin invisible, et de pointer ensuite le contenu de ce contrôle Image :
 - `NomduContrôleGlissé.DragIcon = NomduContrôleImage.Picture`
 - **Etape 3** : il faut préciser ce qui doit se passer lorsque l'utilisateur va relâcher le bouton de la souris au-dessus de tel ou tel contrôle. Ceci est un événement DragDrop. C'est en particulier là qu'on va passer les lignes de code déplaçant effectivement l'objet, si tel est l'effet recherché.

- Résumons-nous. Un cliquer-glisser représente donc une série de trois types d'événements potentiels, qu'il va falloir gérer pour chacun des objets susceptibles de les recevoir...



L'utilisateur enfonce le bouton de la souris sur l'objet qu'il souhaite cliquer-glisser. Le curseur prend alors la forme d'un symbole de l'objet (DragIcon)

Il trébale le symbole de l'objet au-dessus de tel et tel contrôle

Il lâche le symbole de l'objet au-dessus de tel ou tel contrôle

- Avec un peu d'habitude et un minimum de rigueur, on peut parvenir à de très jolis résultats.
- **Dames** est un exemple de Drag & Drop où l'on donne l'illusion à l'utilisateur qu'il déplace un objet avec sa souris. Mais rappelez vous, ce n'est qu'une illusion ! Celle-ci est produite par le fait que l'icône de déplacement est la copie conforme de l'objet... Je vous laisse méditer là-dessus. Quant à **Manège**, il emploie un petit truc algorithmique un brin finaud (mais rien de vraiment méchant).

Partie 7

Compléments de code

- **La gestion des Erreurs**
- On sait que dans une application digne de ce nom, même lorsqu'une erreur se produit (fichier introuvable, mauvaise saisie au clavier, etc.) cette erreur ne donne pas lieu à un plantage inopiné du programme, mais qu'elle est gérée par ce programme. Cette brève section est donc consacrée aux erreurs à l'exécution, et au moyen de les prévoir (à défaut de pouvoir les empêcher).
- **L'objet Err**
- C'est là l'occasion de découvrir, après App, le deuxième objet de ce cours qui ne soit pas un contrôle : l'objet Err.
- Cet objet, qui n'étant pas un contrôle, ne possède pas d'existence graphique, est utilisé par toute application pour stocker l'état des erreurs qu'elle a éventuellement provoquées. On pourra, dans le code, avoir recours à différentes propriétés de l'objet Err :
- Number : désigne le code de cette erreur, permettant d'identifier son type
- Description : il s'agit d'un rapide bla-bla sur la nature de l'erreur
- Source : qui indique l'objet à l'origine de l'erreur

- **Gérer les erreurs**

- Il y a deux choses à savoir avant de pouvoir procéder :
- Toute erreur provoque un événement appelé >Error.
- VB impose de gérer les erreurs par une programmation non structurée. C'est une pure horreur, mais c'est comme ça, on n'y peut rien.
- Il faudra donc, au début d'une procédure susceptible d'engendrer des erreurs (les mauvaises langues traduiront : "au début de chaque procédure...), taper la ligne suivante :
- On Error GoTo Etiquette
- Une étiquette, c'est une sorte de sous-titre à l'intérieur de la procédure, se terminant par le signe ":" (deux points).
- Oui, je sais, c'est une pure horreur que d'obliger les gens à mettre des GoTo dans un joli programme structuré. Mais que voulez-vous, ici, Microsoft ne nous a pas laissé le choix. L'idée est donc que si une erreur, quelle qu'elle soit, survient au cours de cette procédure, au lieu de se planter bêtement, l'application ira gentiment rechercher l'étiquette que vous avez désignée, et exécutera son code.

- Et du coup, pour éviter que le déroulement normal de la procédure n'exécute lui aussi ce qui se trouve sous l'étiquette, il faudra forcer la sortie de la procédure avant cette étiquette par l'instruction Exit Sub. Voilà ce que cela donne :
- ```
Public Sub Totoche()
On Error GoTo Oops
Tata = 12 / 0
Exit Sub
Oops:
Msgbox("Diviser par zéro, c'est pas malin !")
End Sub
```
- Bien sûr, ici, notre gestionnaire d'erreurs est vraiment a minima. On peut faire quelque chose de beaucoup plus fin en y testant par exemple la propriété Err.Number, et en réagissant différemment selon le code (donc le type) de l'erreur qui est survenue.

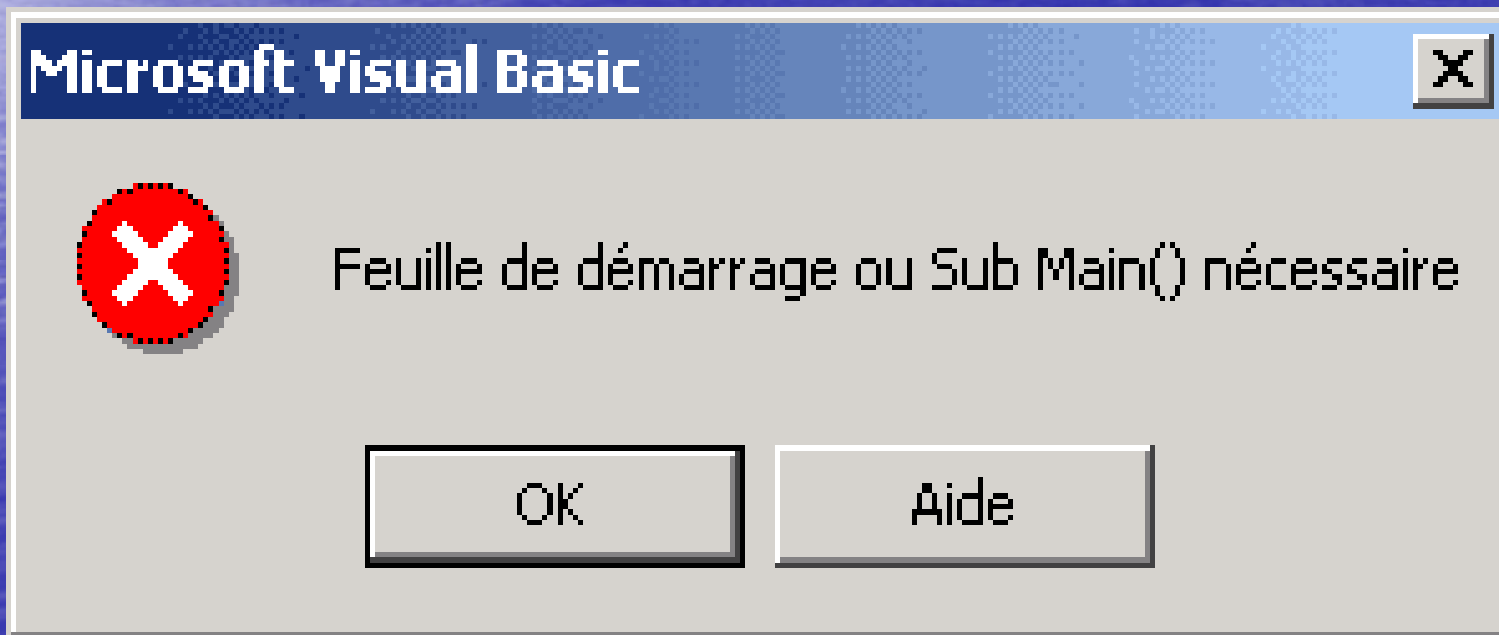
- Fonctions et Procédures personnalisées
- Jusqu'ici, toutes les procédures que nous avons utilisées étaient des procédures événementielles : leur nom était régi par des règles strictes, qui spécifiaient le déclenchement de leur exécution. Mais, nous l'avons dit en introduction, , VB permet comme tout langage de créer des procédures traditionnelles, baptisées selon votre bon vouloir. Leur exécution sera alors déclenchée non par une action de l'utilisateur, mais par une instruction de votre programme. C'est donc l'équivalent VB du Gosub cher aux cœurs de certains (d'ailleurs, les nostalgiques intégristes peuvent toujours utiliser Gosub, qui est une instruction reconnue en VB).
- Je rappelle que de tels appels peuvent comporter des paramètres, ou non, cela ne change rien de fondamental à l'affaire.

- **Un cas banal d'utilisation de cette technique est celui de l'initialisation d'une Form. Pour que cette Form soit propre, on veut que toutes les zones de textes soient vierges. Pour peu qu'on ait besoin de procéder à l'initialisation à plusieurs moments différents de notre application, au lieu de répéter à chaque fois les mêmes codes, on va créer une procédure Init() qui comportera les lignes de code nécessaires :**
- Private Sub Init()  
Nom.Text = ""  
Prénom.Text = ""  
Etc.  
End Sub
- Pour appeler cette procédure, il suffira ensuite de passer l'instruction suivante :
- Call Init
- Et le tour est joué. Quelques éléments supplémentaires toutefois :
- Attention à la portée des procédures. Une procédure Private n'est disponible que si l'appel se situe dans la Form qui la contient. Pour qu'une procédure puisse être appelée de n'importe quelle Form, il faut qu'elle porte comme nom Public. Dans ce cas, lors de l'appel, la procédure désignée doit être désignée comme faisant partie d'une autre Form. Si l'on souhaite appeler dans Form1 une procédure appelée Truc est qui est déclarée dans Form2, il faudra écrire :



- Call Form2.Truc
- Au passage, j'en profite pour signaler que cette règle vaut pour désigner n'importe quel objet qui ne se trouve pas sur la Form à partir de laquelle on le désigne.
- Si l'on veut passer des paramètres à une procédure, on aura recours à deux instructions : ByVal pour le passage par valeur (paramètre en entrée), et ByVal pour le passage par référence. La déclaration d'une procédure Toto avec un paramètre numérique X en entrée et un paramètre Y en sortie donnera ainsi :
- Private Sub Toto(ByVal X as Integer, ByVal Y as Integer)
- Pour déclarer une Fonction, on aura recours aux instructions Function et End Function. Voici une fonction pas très utile, qui retourne bêtement le carré d'un nombre :
- Private Function Carre(X as Double)  
Carre = X \* X  
End Function

- **Procédure de démarrage**
- Lors du lancement de l'application (du projet), VB a besoin de savoir quelle procédure (ou quelle Form), est la procédure (la Form) par défaut. Si vous ne touchez à rien, il s'agit toujours de la première Form qui a été créée dans le projet. Mais en cas de copier-coller sauvages de Forms , VB peut finir par se mélanger les pédales et ne plus savoir par où votre application est censée commencer. Il vous envoie alors un magnifique message d'erreur, le très classique



- Si vous souhaitez changer de Form par défaut, ou la redéfinir en cas d'erreur, ou si vous souhaitez lancer le projet par une procédure principale indépendante de toute Form, cela se passe dans le menu Projet – Propriétés. Vous y choisissez de lancer l'application :
- Soit en redéfinissant la Form par défaut
- Soit en désignant une procédure principale, qui devra obligatoirement se trouver dans un module et s'appeler Sub Main()

## Partie 8

- **Gestion des fichiers**
- On n'abordera pas ici la manière dont on peut attaquer des fichiers complexes (comme des bases de données), via des contrôles ad hoc. Ce point a été délibérément zappé du cours, pour des questions de temps et de redondances avec d'autres matières.
- Ce petit chapitre a pour but plus limité, de vous fournir un outil simple et universel pour stocker et récupérer des données d'une exécution à une autre, au travers d'un genre de fichier qu'on a déjà dû vous présenter amplement en C : le fichier texte à accès séquentiel.
- On peut également accéder aux fichiers en "accès direct" ; il existe aussi des fichiers dits "binaires", mais il faut faire vite, et on se concentrera ici sur le type le plus basique (mais pas le moins utile).
- Pour plus de précisions, il n'est pas interdit de zieuter attentivement sur la [gestion de fichiers](#) un excellent [cours d'algorithmique](#) que je recommande vivement à ceux qui ne le connaissent pas..

- **Ouvrir et fermer un fichier**
- Pour ouvrir un fichier, vous devez connaître son nom (y compris le chemin d'accès), dire ce que vous voulez en faire (lire, écrire, ou ajouter ?) et lui attribuer un numéro arbitraire, dit numéro de canal. Cela donnera ainsi :
- `Open "C:\TextFile.txt" For Input As #1`
- Bien sûr, si vous utilisez plusieurs fichiers simultanément, il est impératif de leur attribuer chacun un numéro de canal différent... `For Input` signifie qu'on ouvre le fichier en lecture. Pour l'ouvrir en écriture, on utilisera `For Output`, et pour l'ouvrir en ajout, `For Append`.
- **Rappel** : à l'ouverture, il faut choisir, on ne peut faire qu'une seule chose à la fois avec un fichier.
- Et puis tant qu'on y est, et d'un même élan sublime, voici l'instruction de fermeture d'un fichier :
- `Close #i`
- où `i` est le numéro de canal du fichier à fermer.

- Lire un fichier
- Il y a trois possibilités. Deux que je ne cite que pour l'anecdote, car mieux vaut les éviter, et une vraiment utile.
- Pour l'anecdote uniquement, donc :
- NomDeVariable = Input (100, #i)
- ...recopie dans NomDeVariable les 100 caractères suivants du fichier numéro i, à partir de la dernière lecture. Sinon, on peut aussi faire :
- Toto = LOF(#i)  
Input (Toto, #i)
- Toto récupère ici le nombre de caractères du fichier texte, via la fonction LOF. On recopie ensuite tout le fichier d'un seul coup d'un seul dans la variable NomDeVariable.
- Mais l'avantage des fichiers texte étant d'être organisés par lignes, c'est-à-dire par enregistrements, voilà ce qu'il faut utiliser afin de ne pas perdre cette structuration :
- Line Input #i, NomDeVariable

- C'est de loin la meilleure solution. Elle suppose toutefois que le fichier texte soit correctement construit, c'est-à-dire que chaque série de champs se termine par les caractères CR LF, formant ainsi une ligne (vous n'avez pas besoin de vous tracasser pour cela, si vous avez vous-mêmes correctement créé le fichier texte). A ce moment-là, chaque lecture avec Line Input copie la ligne suivante du fichier dans NomDeVariable.
- Si vous avez eu de surcroît la bonne idée de bâtir votre fichier sous forme de champs de largeur fixe, ce sera un jeu d'enfants de récupérer, au sein de chaque ligne, les différents champs. D'une manière ou d'une autre, il suffira de découper à un moment ou à un autre NomDeVariable en différents morceaux via la fonction Mid (on aura soin d'épurer les espaces inutiles grâce à la fonction Trim mentionnée précédemment.

- Je rappelle qu'avec les fichiers, la technique standard consiste à balancer d'entrée de jeu l'intégralité du fichier en mémoire vive, autrement dit dans un tableau (on évite de faire cela uniquement si le fichier est vraiment trop massif pour tenir en mémoire, ce qui est un cas très rare). En revanche, pour le détail, on peut opter pour plusieurs options : soit "casser" d'entrée chaque ligne du fichier en différents champs qui rempliront différents tableaux, soit tout envoyer dans un seul tableau, à raison d'une ligne complète par élément du tableau, ligne dont on extrait ensuite les champs au fur et à mesure des besoins. C'est au choix du client. Cette seconde technique donnera, extrêmement classiquement, les lignes de code suivantes :

- Dim T() as string

```
...
Open "C:\Monfichier.txt" As #1 For Input
i = -1
While Not Eof(1)
 i = i + 1
 Redim Preserve T(i)
 Line Input #1, T(i)
Wend
```

- Le principe est simple : on crée un tableau dynamique. On parcourt le fichier ligne après ligne tant qu'il en reste (boucle While ... Wend). A chaque ligne, on actualise l'indice i, on redimensionne le tableau, et on recopie la ligne dans la nouvelle case ainsi créée, tout en préservant les cases précédemment remplies. Et hop.



- Ecrire dans un fichier

- Là, c'est carrément du gâteau. L'écriture se fait systématiquement ligne par ligne. L'instruction à utiliser est :

- Print #i, NomDeVariable

- ...qui écrit donc le contenu de la variable NomDeVariable dans le fichier de numéro i. Encore une fois, pour être certain que les différentes informations sont rangées en respectant les champs de largeur fixe, le plus simple aura été de déclarer certaines variables String comme possédant toujours un nombre fixe de caractères.

- Par exemple, en ayant déclaré :

- Dim Nom as String\*25

- Je le répète, j'ai créé une variable caractère Nom qui, quoi qu'il arrive dans la suite du code, possédera toujours 25 caractères exactement. Je constituerai donc une ligne du fichier (la variable caractère NomDeVariable) par concaténation de différentes variables caractères de largeur fixe, et j'enverrai ensuite cette ligne dans le fichier par un Print. Ainsi, on est certain de respecter la structure du fichier et de pouvoir réutiliser celui-ci la fois suivante.

- Si l'on a choisi la meilleure solution, à savoir utiliser les variables structurées, il reste tout de même un petit souci. C'est que si VB autorise la lecture d'une ligne dans une variable structurée, il interdit la recopie directe d'une variable structurée dans un fichier texte. C'est énervant, mais c'est comme ça. Cette recopie devra donc se faire sous forme de concaténation de champs, par exemple :

- Print #2, T(i).Nom & T(i).Prenom & T(i).Tel
- Allez, vous en savez maintenant bien assez. Roulez jeunesse :
- Attention ! Pour fonctionner, ces applications ont besoin que le(s) fichier(s) texte associé(s) aient été téléchargés et aient été placés dans le même répertoire que l'exécutable

## Partie 9

- **Distribuer une application**
- **Attention aux images !**
- Ce qui suit est de la plus haute importance ! (ce qui précédait aussi, d'ailleurs).
- Lorsqu'on a recours dans une application VB à un fichier image (du type .bmp), par exemple, on pourrait penser qu'il suffit de poser un contrôle Image, et de définir sa propriété Picture (son contenu) en désignant l'image voulue. Malheureusement, ce n'est pas si simple. A l'exécution, il ne suffira pas en effet qu'on ait dit qu'on voulait tel fichier image dans tel contrôle. Il faudra aussi impérativement que ce fichier ait été préalablement chargé en mémoire.
- Dès lors, il y a deux stratégies possibles.

- **on inclut directement les fichiers image dans l'application. C'est-à-dire que la propriété Picture du contrôle est réglée par défaut sur le fichier image à inclure. La conséquence est que lors de la compilation, VB repère que tel ou tel fichier image doit être inscrit dans un contrôle. Du coup, il embarquera in extenso le fichier image au sein de l'exécutable. Avantage, les graphiques feront partie à part entière de cet exécutable, et on est certain qu'ils seront présents quand on en aura besoin. Inconvénient, cela alourdit l'exécutable, parfois dans des proportions désastreuses. Moralité, on ne doit employer cette technique commode que pour les petits fichiers**

- images, ne pesant pas lourd en termes de mémoire. C'est d'ailleurs ce que je me suis permis de faire pour les exercices livrés à votre pâture dans ce cours.
- l'autre stratégie consiste à ne pas affecter par défaut la propriété Picture des contrôles, et on utilise la fonction LoadPicture dans le code. Cette fonction a pour rôle de charger au cours de l'exécution l'image dans le contrôle, en allant chercher le fichier adéquat sur le disque dur, le CD-ROM, etc. Dès lors, l'image ne sera pas incluse dans l'exécutable. Celui-ci se contentera, lorsqu'il tombera sur cette instruction, d'aller chercher le fichier pointé par cette ligne de code.
- Cette dernière technique est certes un plus difficile à mettre en oeuvre, mais c'est la seule qui assure que l'exécutable restera dans des proportions raisonnables.
- Elle implique deux choses :
- Premièrement, il faut écrire le code en pointant correctement sur les fichiers images qui devront être situés dans un répertoire relatif à celui de l'application. Couramment, quel que soit le répertoire de l'exécutable, on constitue par exemple un sous-répertoire "Images", dans lequel on stocke tous les fichiers images dont l'application a besoin. Le chargement des images s'accomplira via le code suivant (on suppose que le contrôle image s'appelle Tableau, et que le fichier est "VanGogh.jpg" :

- `Chemin = App.Path`  
`If Right(Chemin, 1) <> "\" Then`  
    `Chemin = Chemin & "\"`  
`Endif`  
`Chemin = Chemin & "Images\"`  
`Tableau.Picture = LoadPicture(Chemin & "VGogh.jpg")`
- Mais, et c'est le deuxième point, pour que cette affaire continue à marcher lorsque je vais distribuer mon application sur d'autres machines, il faut que je puisse être certain que les fichiers images seront bien présents dans le bon répertoire, celui que pointe l'instruction `LoadPicture` (en l'occurrence, dans un sous répertoire "Images" de mon application. C'est entre autres à cela que sert un indispensable utilitaire, l'installateur automatique.

## Utiliser l'installateur automatique

- Votre application est enfin terminée. Elle est blonde, elle est belle, elle sent bon le sable chaud. Elle tourne sans erreurs et sans même remuer les oreilles. Bref, vous pouvez être fiers de vous.
- Il ne reste plus qu'à faire profiter le reste de l'Humanité de votre Œuvre ; car ce serait trop dommage de l'en priver.
- Commençons par le début : vous allez compiler le projet, créant ainsi un fichier \*.exe, qui sera dorénavant repéré par une jolie icône (qui n'est autre que l'icône de la Form par défaut du projet). Evitez de laisser l'icône par défaut, il n'y a rien qui fasse davantage toc d'entrée de jeu.
- Mais là, restent quelques détails à régler. En effet, si vous ne transmettez à quelqu'un que ce fichier exe, il risque de ne rien pouvoir en faire du tout, car peuvent vraisemblablement lui manquer :

- les bibliothèques de VB, à savoir des fichiers de type DLL auxquels l'exécutable devra s'adresser en permanence pour effectuer son travail. Sans ces bibliothèques, l'application refusera de tourner. Eh oui, un exécutable VB n'est jamais réellement autonome.
- l'ensemble des fichiers "extérieurs" mais néanmoins indispensables à votre application, tels que les fichiers images vus il y a un instant, des éventuels fichiers texte, ou des sons, ou que sais-je encore, bref, tous les fichiers auxquels votre exécutable fera appel à un moment où à un autre, parce vous l'avez décidé.
- S'il manque l'un ou l'autre de ces ingrédients, votre exécutable tournera dans le vide durant une infime fraction de seconde et se plantera avec un long ululement d'agonie. Terrifiant spectacle.
- C'est pourquoi un logiciel spécial est prévu, l'assistant d'installation, qui va se charger de produire à partir de votre application un fichier apte à s'installer de manière propre sur une autre machine.
- Il n'y a pour ainsi dire qu'à suivre les instructions.



- Etape 1 :sélection du fichier projet
- Etape 2 :choix du mode de déploiement (a priori, standard, "dossier unique")
- Etape 3 :choix du lieu d'empaquetage (n'importe quel répertoire vierge fera l'affaire). Il s'agit du répertoire de votre machine dans lequel l'assistant va écrire le résultat de son travail.
- Etape 4 :composants ActiveX : ignorez cette étape, sauf si vous avez inclus dans votre application des composants exotiques (voir [partie 11](#)).
- Etape 5 :fichiers à inclure. C'est l'étape décisive. A priori, vous devez conserver tous les fichiers déjà proposés par l'assistant (dont les fameuses DLL nécessaires, qu'il repère tout seul). Mais surtout, c'est à cette étape que vous pouvez en ajouter d'autres, en particulier ces fichiers image, fichiers textes, etc. dont vous aurez auparavant dressé la liste complète, car l'assistant ne le fera pas pour vous). Cliquez sur "Ajouter" pour inclure tous ces autres fichiers nécessaires dans le paquet cadeau à fabriquer. Et précisez bien dans quel répertoire de destination ces fichiers devront être installés.
- Le résultat de tout cela est un fichier auto-extractible (de type Setup) qui produira de merveilleuses installations Windows de type tout à fait professionnelles, et surtout, qui s'exécuteront sans erreurs.

# Partie 10

- **Les contrôles à la chaîne**
- **La gestion dynamique des contrôles**
- Jusqu'à présent, lorsque nous voulions utiliser des contrôles dans une application, nous avons toujours procédé de la même manière : au départ, on crée tous les contrôles nécessaires à un moment ou à un autre dans l'application, quitte à en cacher provisoirement quelques uns hors de la vue de l'utilisateur. Et ensuite, on permet à l'utilisateur d'effectuer certaines actions sur ces contrôles.
- Cette stratégie, dans 99 % des cas, donne des résultats tout à fait convenables. Mais pour le 1% restant, il est bon de connaître une autre manière de s'y prendre : celle qui va nous permettre de créer et de faire disparaître les contrôles au cours de l'exécution de l'application.
- Un exemple de ce 1%, nous est donné par un jeu comme le démineur, que tout le monde connaît pour l'avoir pratiqué de longues heures durant, à l'insu de son chef de bureau. Au démineur, les règles du jeu sont invariables. Mais en revanche, l'utilisateur a le droit de choisir entre plusieurs tailles de damier. Pire, il peut même choisir librement le nombre de cases en ligne et colonne.

- Une manière barbare de programmer cela, serait de créer au départ le plus grand damier autorisé pour le jeu ; puis ensuite, à la lumière du choix effectué par le joueur, de masquer les cases qui n'ont pas lieu d'être. Mais c'est une solution d'une part bien peu élégante (et, question philosophique, l'élégance n'est-elle pas la marque de tout bon programmeur ?), d'autre part très lourde en termes de ressources mémoire, puisqu'on va mobiliser de la place pour gérer des dizaines de contrôles qui s'avèreront le plus souvent aussi superflus que le peigne de Fabien Barthez.
- Il convient donc de posséder la technique permettant de créer, de manipuler et de détruire des contrôles par des instructions figurant dans l'application elle-même. Ce n'est pas vraiment difficile, et en plus, c'est une excellente manière de se préparer à la programmation objet proprement dite.
- Alors, comment procéder ? Réfléchissons un peu, ça ne pourra pas nous faire de mal.
- Si nous créons un contrôle de toutes pièces au cours de l'exécution, et si l'on veut que ce contrôle puisse servir à quelque chose, il va bien falloir qu'il existe les procédures d'événements qui lui sont liées. Or, on n'a aucun moyen de créer en cours de route lesdites procédures (un programme ne peut pas écrire des lignes de programme). Ainsi, si l'on veut que notre contrôle soit autre chose qu'un simple élément décoratif, il faut que les procédures événementielles qui s'y rapportent aient été créées à l'avance.

- Comment est-ce possible ? Tout simplement en ne créant dynamiquement que des éléments de groupes, dont on aura préalablement défini le premier élément et les procédures associées.
- Quatre-vingt-dix-neuf fois sur cent, il faut donc appliquer la stratégie suivante :
- On crée à la main l'élément numéro zéro d'un groupe de contrôles (quitte à le masquer provisoirement), et on définit ses propriétés.
- On écrit les procédures événementielles liées à ce groupe
- On engendre de manière dynamique les autres éléments du groupe, dans la quantité souhaitée.
- Pour créer un élément supplémentaire d'un groupe de contrôles, on emploiera le code suivant :
- `Load NomduGroupe(i)`
- où "NomduContrôle" est évidemment le nom du groupe, et "i" l'index de l'élément qui sera créé.
- Remarque importante : tout nouvel élément, créé par Load, d'un groupe, est situé par défaut à l'emplacement exact de l'original, et invisible. Il faudra donc le plus souvent modifier ses propriétés Top et Left, ainsi que sa propriété Visible.
- Pour supprimer un élément d'un groupe de contrôles, on écrira :
- `Unload NomduGroupe(i)`

- C'est aussi simple que cela ! Pour peu qu'on ne fasse pas n'importe quoi, la création et la destruction dynamiques de contrôles ne posent donc pas la moindre difficulté. En fin de compte, pour conclure sur ce point : même si le cheminement diffère un peu, les instructions Load et Unload nous permettent de parvenir au même résultat avec les groupes de contrôles que l'instruction Redim avec les tableaux de variables.

- La notion de collection

- Rien à voir avec la haute couture de saison, voici pour conclure un concept fort utile dès que l'on est amené à gérer à la queue leu leu des hordes sauvages de contrôles.
- Dans cet esprit, on disposait déjà d'un outil simple, mais très performant : les groupes
- , qui comme le disait le sage, à savoir moi-même, sont aux contrôles ce que les tableaux sont aux variables. Eh bien, on pourrait dire, pour filer la métaphore, que les collections sont aux contrôles ce que les variables structurées sont aux variables : un assemblage d'éléments au besoin hétéroclite, permettant un traitement global là où il fallait auparavant prendre les éléments un par un.
- Première bonne nouvelle : tous les contrôles posés sur une feuille appartiennent de droit à la collection de la feuille. Au sein de cette collection, exactement comme dans un groupe, chaque élément est désigné par un numéro d'indice (correspondant ici à l'ordre de création de l'élément dans la feuille).
- Pour accéder aux éléments de la collection, on pourra employer la propriété Controls de la Form, par exemple de la manière suivante :
- `Form1.Controls(5).Visible = False`
- ...qui masquera le sixième contrôle créé sur Form1.

- A noter que la propriété Controls n'est pas accessible depuis la fenêtre des propriétés du mode conception, mais uniquement par le code.
- Evidemment, prendre les contrôles un par un par leur numéro d'indice, c'est bien, mais le problème se pose vite de savoir à quel numéro s'arrêter (sinon, vlan, c'est le dépassement d'indice direct garanti sur facture). Pour cela, deux solutions :
- La propriété Count de la collection Controls, qui dénombre le nombre de contrôles qu'elle contient. Pour rendre visibles tous les contrôles d'une feuille, nous pourrions donc écrire :
- ```
For i = 0 to Form1.Controls.Count - 1  
    Form1.Controls(i).Visible = True  
Next i
```
- Mais il y a une autre ruse, bien plus redoutable...
- C'est qu'en VB, on dispose d'une instruction de boucle spéciale pour les collections de contrôles. Il s'agit de l'instruction
- For Each Truc in Machin
- où "Machin" est la collection de contrôles (donc Form1.Controls, par exemple), et "Truc" est une variable qui prendra successivement la valeur de chaque contrôle de la collection. Une telle variable, qui ne contient pas une valeur mais un objet, est appelée... une variable objet. Etonnant, non ? Ce qui nous permettra d'écrire une boucle aussi simple que :

- For Each Bidule In Form1.Controls
 Bidule.Visible = True
Next Bidule
- Quelle que soit la méthode utilisée, parcourir automatiquement toute une série, c'est bien, mais pouvoir déterminer à quel type de contrôle on a affaire, ce serait nettement mieux. Car pour certains traitements, on a besoin de savoir si le contrôle est un bouton, une zone de texte ou tout autre chose avant de pouvoir lui infliger quoi que ce soit.
- Eh bien, une fois de plus, VB a tout prévu, par l'instruction :
- Typeof Bidule Is Nomtype
- Où "Bidule" est un contrôle (ou une variable objet), et "Nomtype" un mot-clé désignant un type (une classe) de contrôles :
 CommandButton, TextBox, ListBox, etc.
- Pour conclure, imaginons un fragment de code verrouillant toutes les zones de texte d'une feuille, sans que ces zones de texte aient nécessairement été créées comme un groupe de contrôle :
- For Each Machin In Form1.Controls
 If Machin Is TextBox Then
 Machin.Locked = True
 Endif
Next Machin
- Et le tour est joué. Puissant, n'est-il pas ?

- Insérer des menus
- Une application Windows digne de ce nom est fréquemment pilotée par des menus. Avant d'aller plus loin, examinons quelques points de vocabulaire :

The image shows a screenshot of a Windows application menu bar with the following items: Fichier, Edition, Image, Calque, Sélection, Filtre, Affichage, Fenêtre, Aide. The 'Affichage' menu is open, displaying the following items:

- Nouvelle vue
- Format d'épreuve ▶
- Couleurs de l'épreuve Ctrl+Y
- Couleurs non imprimables Maj+Ctrl+Y
- Agrandir Ctrl++
- Réduire Ctrl+-
- Taille écran Ctrl+\$
- Taille réelle des pixels Alt+Ctrl+0
- Taille d'impression
- ✓ Afficher les extras Ctrl+H
- Afficher ▶
- Afficher les règles Ctrl+R
- ✓ Magnétisme Maj+Ctrl+ù
- Magnétisme ▶
- Verrouiller les repères Alt+Ctrl+
- Effacer les repères
- Nouveau repère...
- Verrouiller les tranches
- Effacer les tranches

Annotations with green boxes and arrows point to the following elements:

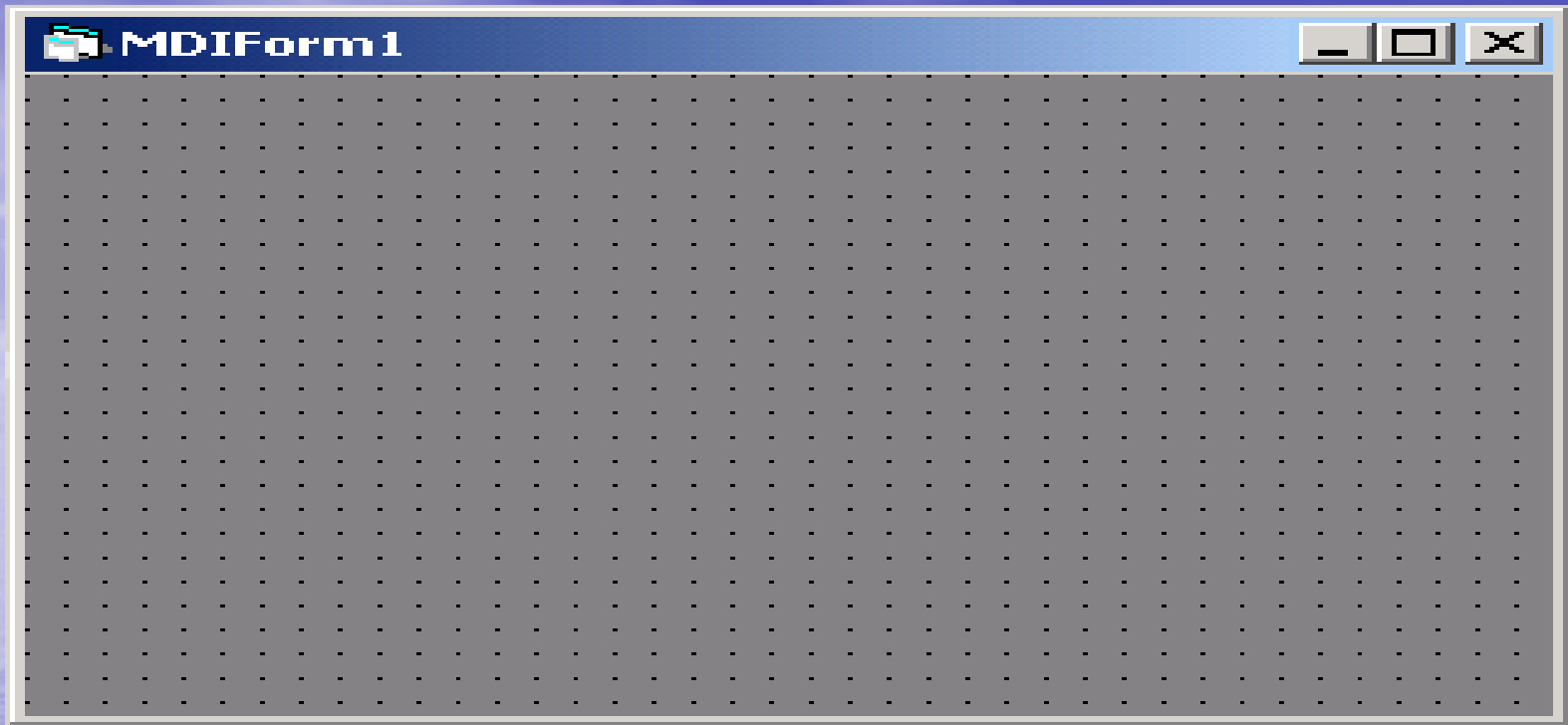
- Menu déroulant**: Points to the 'Affichage' menu title.
- Élément de menu**: Points to the 'Agrandir' menu item.
- Séparateur**: Points to a horizontal line separating menu sections.
- Sous-menu**: Points to the 'Magnétisme' menu item, which has a submenu arrow.
- Menu inactivé**: Points to the 'Effacer les repères' menu item, which is dimmed.

- Concevoir des menus pour une application VB est d'une simplicité confondante, dans la mesure où il existe un outil spécial : le créateur de menus, accessible par la commande Outils - Créateur de Menus. Ensuite, c'est une affaire de quelques minutes de prise en main.
- Cet outil fait apparaître l'ensemble de la structure des menus d'une Form donnée, et nous permet de la gérer. Le seul point à comprendre est que chaque élément de menu, pour VB, est un objet, et à ce titre possède donc un certain nombre de propriétés :
- Caption : le texte du menu, tel qu'il apparaît à l'écran (ex : Fichier, Edition, Affichage, etc.)
- Name : le nom de cet élément de menu pour l'application.
- Checked : indique si l'élément de menu est coché par un petit "v" à sa gauche
- Enabled : indique si l'élément de menu est actif ou inactif
- Dans la liste des menus, les niveaux des retraits indiquent le niveau des menus. Les séparateurs correspondent tout simplement à des éléments qui ont un Name, mais dont le Caption est un simple trait d'union.
- Ensuite, chaque élément de menu, qui je le rappelle, est un objet, va pouvoir déclencher un traitement par la programmation de la procédure qui lui sera associée :
- ```
Private Sub NameElementMenu_Click()
...
End Sub
```

- VB permet également de programmer des menus dits pop-up ou contextuels. En deux mots : après avoir créé le menu par le créateur de menus, il suffit en quelque sorte de l'appeler par le code adéquat, en l'occurrence :
- `PopupMenu NameDuMenu, ConstanteVb`
- Cette instruction `PopupMenu` sera le plus souvent placée dans une procédure `MouseDown`, après un test vérifiant que c'est bien le bouton droit qui a été enfoncé. Mais on peut imaginer des menus contextuels apparaissant à d'autres occasions. Quant à la constante `vb` qui conclut l'instruction, elle indique le positionnement du menu par rapport à la souris (voir l'aide VB sur ce point précis).
- Enfin, pour qu'un menu soit exclusivement contextuel et n'apparaisse pas en permanence dans la barre de menus du haut de la Form, il suffit de mettre dans le créateur de menus sa propriété `Visible` à `False`, et le tour est joué.
- Le bonheur, cela tient parfois à peu de choses.

- Les interfaces à Form multiples (MDI)
- Nous voilà armés pour faire sauter une nouvelle limite dans VB : celle de la form unique.
- En effet, jusqu'à présent, nos applications ne comportaient qu'une Form et une seule à la fois. Même si nous disposions de plusieurs Form, celles-ci n'étaient disponibles pour l'utilisateur que successivement : le code cachait une Form, en rendait une autre visible, et c'était tout ce que l'on pouvait faire.
- Or, dans Windows, la plupart des applications utilisent plusieurs Form en même temps, ou tout au moins laissent cette possibilité ouverte. Qu'on pense à Word : il y a la fenêtre principale, celle de Word lui-même. Mais ensuite, au sein de cette fenêtre, l'utilisateur peut ouvrir autant de documents qu'il le souhaite, et chacun de ces documents apparaîtra dans une nouvelle fenêtre.
- Il est donc grand temps que nous apprenions à utiliser cette possibilité avec VB.

- Une application VB gérant des fenêtres multiples est dite application M.D.I. (pour Multiple Document Interface). Dans une telle application, il y a obligatoirement une Form mère (et une seule) et un certain nombre de Form filles, contenues dans la Form mère.
- Tout ceci peut être créé à la main, ou par du code... et vous comprenez pourquoi nous abordons ceci après avoir vu les instructions Load, Unload et la notion de Collection.
- Création de la Form MDI
- Commençons par la création à la main de la Form mère. On choisira dans le menu adéquat une nouvelle Form, mais d'un type particulier : la Form MDI. Celle-ci apparaît avec un fond spécial, plus sombre qu'une Form classique :



- Une Form MDI ne servant que de conteneur aux Form filles, on ne peut pas poser dessus n'importe quels contrôles. En fait, une telle Form ne peut contenir qu'un nombre très limité de choses :
- un Timer
- une ImageBox (qui sera alors automatiquement de la largeur de la Form elle-même)
- quelques autres contrôles exotiques que nous ne connaissons pas encore...
- des barres d'outils
- des menus
- Si ce site n'explique pas comment créer des barres d'outils, on vient en revanche de voir comment créer des menus. Donc, pas de problèmes.
- Pour finir sur les Form mères, remarquons qu'il ne peut y en avoir qu'une et une seule par projet VB. Si vous essayez d'en créer une deuxième, VB vous l'interdira aussi sec. Logique.
- Passons maintenant aux Form filles. Une Form fille est une Form tout ce qu'il y a de plus normal. Elle a simplement une caractéristique, c'est que sa propriété MDIChild vaut True. C'est cela qui indique à l'application que cette Form doit apparaître au sein de la Form mère de l'application. Il peut bien entendu y avoir autant de Form filles que l'on souhaite.
- Cette histoire de mère et de filles, c'est donc vraiment... un jeu d'enfant (celle-là, je ne pouvais décemment pas passer à côté).

- Pour qu'à l'exécution, une Form fille apparaisse dans la Form mère, il suffit que la Form fille soit l'objet d'une instruction Load. Pour la faire disparaître, évidemment, l'instruction Unload s'impose.
- **Créer les Form filles par du code**
- Une Form étant un objet comme n'importe quel autre, on peut lui appliquer les méthodes de création dynamiques vues au début de ce chapitre. Malheureusement, il n'est pas possible de créer un groupe de Form, comme nous le faisons pour les autres contrôles. Alors, serions-nous coincés ?
- Que nenni. Si la première Form de notre application s'appelle FormIntérieure (propriété Name), on pourra taper le code suivant pour créer une nouvelle Form (par exemple, lorsque l'utilisateur clique sur un éventuel menu intitulé Nouveau) :
- **Dim Toto As New FormIntérieure  
Toto.Visible = True**



- La première ligne crée une copie de l'objet FormIntérieure (on parle en jargon objet de nouvelle instance), qui possède en tout point les mêmes propriétés, les mêmes contrôles, etc. Ce nouvel objet, copie conforme de FormIntérieure, est stocké dans une variable objet qui permet de le désigner, en l'occurrence Toto.
- Petit souci : toutes les nouvelles Form ainsi créés vont être successivement désignées par la variable Toto. Et comme elles ne forment pas un groupe, on est a priori bien en peine de les différencier.
- A cela, il existe deux parades simples, efficaces et de bon goût.
- Lorsqu'il s'agit de traiter toutes les Form d'une application, on pourra utiliser la notion de collection vue plus haut, et écrire une boucle adéquate :
- For Each truc In Forms  
truc.BackColor = vbWhite  
Next truc
- ...La collection Forms nous permettant de balayer toutes les Forms de l'application une par une.

- Enfin, dernier problème abordé ici, lorsque nous avons besoin de localiser un contrôle d'une application MDI, nous ne savons pas forcément de quelle Form est issu ce contrôle. Imaginons que nous voulions, à un moment donné, changer ce qu'il y a écrit sur le bouton de commande Bouton1. On va alors être obligé, faute de provoquer une erreur, de préciser qu'il s'agit du contrôle Bouton1 de la Form fille active, par opposition aux autres boutons Bouton1 des Form filles non actives). Car à ce moment là de l'application, il y a autant de Bouton1 que de Form filles dans l'application ! Pour parler du Bouton1 de la Form active, rien de plus simple :
- `ActiveForm.Bouton1.Caption = "coucou"`
- Vous voyez, pour conclure, que VB nous ouvre assez facilement les portes de la réalisation d'applications au look vraiment professionnel. Après, plus ça grossit, plus il faut être méthodique pour ne pas s'y perdre...

# Partie 11

## Attaquer (sauvagement) les tréfonds de Windows:

- **dll, api, ocx... et sos !**
- Gare à la casse et sus à la migraine ! Dans cette rapide présentation, on va voir quelques (je dis bien quelques) éléments permettant d'aller plus loin avec VB. En fait, on va voir comment VB peut (mais ce n'est pas obligé) utiliser des éléments de Windows pour améliorer ses (vos) performances. Mais là, inutile de préciser qu'entrant de plain-pied dans le domaine ésotérique de l'architecture de Windows, on n'est pas au bout de nos peines.
- **Le rôle des DLL**
- Vous savez sans doute (ne serait-ce que pour avoir supprimé un jour par mégarde) que ces fameuses DLL jouent un rôle capital dans Windows. Mais que se cache-t-il derrière cet acronyme byzantin ? Eh bien, ce sont des "bibliothèques de liens dynamiques", autrement dit : des Dynamic Link Libraries. Ces DLL contiennent du code compilé (donc illisible) exécutant telles ou telles fonctions dans Windows.

- Par exemple, vous avez déjà remarqué que dans Windows, certaines boîtes de dialogue ont une tête standard, toujours la même quelle que soit l'application : Fichier – Ouvrir, Fichier - Enregistrer, Fichier – Imprimer, Format - Police, et quelques autres encore. Ce n'est pas un hasard : ces boîtes sont en quelque sorte préprogrammées, et n'importe quel programmeur (dont vous-mêmes, public adoré), au lieu de se fader de les reproduire avec plus ou moins de bonheur, peut se contenter d'y faire appel.
- En fait, quel que soit le logiciel de départ, lorsqu'on déclenche par exemple la commande Fichier - Imprimer, c'est toujours le même morceau de code externe à ce logiciel qui s'exécute .
- Pour continuer avec notre exemple, Comdlg32.dll est le doux nom de la DLL qui contient ce code nécessaire à l'exécution de boîtes de dialogue communes (d'où son titre, abréviation de "Common Dialog") aux applications Windows qui veulent s'en servir.

- Donc, lorsque Word ou Excel vous proposent ces boîtes de dialogue, il faut comprendre que les lignes de code correspondantes ne font pas à proprement partie de Word ou Excel. Ces lignes de code sont stockées (une seule fois, donc) dans cette DLL. Word et Excel, eux, se contentent d'appeler la DLL, autrement dit le code qu'elle contient.
- Il en va de même pour toutes les DLL, qui sont donc des morceaux de programme utilisables par d'autres programmes. Cela signifie que lorsque vous écrivez du VB, vous n'êtes pas obligé de réinventer à chaque fois l'eau tiède, la poudre à maquiller et le fil à couper le roquefort. Si une DLL contient déjà le code qui fait ce que vous souhaitez, vous pouvez (c'est même recommandé) appeler cette DLL plutôt que réécrire – généralement, moins bien – le code en question en VB.
- Avec VB, vous pouvez donc :

- utiliser du code déjà présent dans Windows via une DLL
- et même, créer de surcroît vos propres DLL (mais, là, attention les yeux, ça dépasse nettement les objectifs de ce cours.)
- Vous l'aurez compris, on se contentera du premier point (quoique le second ne soit pas si inabordable que cela).
- Alors, au fait, pourquoi « lien dynamique » ? C'est une question d'architecture. Un langage qui fonctionne avec des liens statiques incorpore à l'exécutable absolument tous les éléments de code dont il a besoin (cf. ce qu'on a dit, par exemple, pour le chargement des fichiers dans un contrôle image). Avantage, l'exécutable est autonome, et portable tel quel sur n'importe quel machine. Inconvénient, l'exécutable en question a tendance à être gros. L'avantage des langages et des systèmes fonctionnant avec des liens dynamiques, c'est que le code souvent utilisé est en quelque sorte partagé entre tous les exécutables qui en ont besoin à un moment où à un autre.

- **Avantage** : s'il y a plein d'exécutables, on finit par y gagner en place occupée sur l'ordinateur. Inconvénient, l'exécutable généré par un tel langage ne fonctionnera que s'il trouve présentes sur la machine toutes les DLL dont il a besoin. C'est précisément un des aspects cruciaux dont s'occupe l'assistant d'installation de Visual Basic.
- En fait, avec cette histoire de DLL, on retrouve peu ou prou à un autre niveau la division entre « logiciel d'application » et « système d'exploitation » qui existe depuis si longtemps en informatique. Mais là, la zone d'influence du « système d'exploitation », disons plutôt de Windows, dépasse largement la gestion du disque dur et de l'écran. Les DLL contiennent du code gérant (liste non exhaustive) : l'interface graphique, les capacités multimédia...

- Deux voies vers le bonheur suprême
- VB offre en fait deux possibilités pour appeler le code contenu dans les DLL de Windows. Si l'une d'elle vous est encore inconnue, l'autre est en réalité une vieille connaissance... car tels M. Jourdain, vous n'avez jusqu'à maintenant fait qu'appeler des DLL sans le savoir.
- 2.1 Les appels de l'API
- Commençons par le plus barbare, le plus brutal et le plus sanguinaire. Aïe, Aïe, Aïe, que se cache-t-il derrière API, ce joli nom de pomme rouge ? Comme vous vous en doutez, comme à chaque fois que Microsoft est dans le secteur, ze worm is in ze froot.
- API signifie : Application Programming Interface. C'est en fait un habillage de toutes les fonctions disponibles au sein des DLL. Mais on verra qu'en fait d'habillage, cela ressemble davantage à des haillons qu'à autre chose...
- L'idée générale de l'API, c'est que toute DLL peut être utilisée par un langage comme VB sous forme de fonction, qui utilise donc les paramètres (arguments) qu'on lui fournit et qui renvoie donc un résultat vers le langage. Donc, en théorie, pour utiliser une DLL, il suffit de savoir laquelle effectue le traitement que l'on souhaite, quels arguments il faut lui envoyer, et quel type de valeur elle retourne. Ensuite, il ne reste plus qu'à créer cette fonction dans le programme. Mais... il y a un mais. Et pas qu'un seul !



- Première bonne blague : à part d'épais et indigestes bouquins (en plus, ils sont chers), il n'existe nulle part une liste un peu complète des fonctions d'appel de l'API, vous disant quelle fonction fait quoi, et la syntaxe de l'appel. Donc, en gros, ça existe, mais Microsoft ne vous dit pas ni où ni comment. Résultat, il faut chercher, chercher, et chercher encore, crier au secours sur Internet, pour savoir quel appel API utiliser pour une tâche donnée, et en connaître la syntaxe. Il n'y a pas si longtemps, j'ai vu un étudiant passer plusieurs heures à chercher comment, par un code VB, lancer Access avec une base de données précise chargée à l'ouverture... Ce n'est pourtant pas sorcier, me direz-vous ? Pourtant, il n'a obtenu une réponse valable que juste avant de mourir d'épuisement et de désespoir.

- Mais jetons un voile pudique sur les affres de cette quête, et imaginons que vous ayez enfin trouvé la syntaxe de l'appel API de vos rêves les plus fous. Comment le mettre en œuvre dans votre application ?
- En ce qui concerne la syntaxe, un appel API ne se distingue pas fondamentalement d'un appel à une fonction ordinaire, avec des paramètres.
- Pour pouvoir utiliser de l'API (autrement dit une DLL), il faut tout d'abord déclarer la fonction en question, en indiquant les éléments suivants :
  - le nom de la procédure ou fonction
  - le fichier DLL dans lequel elle se trouve
  - les paramètres qu'elle doit recevoir
  - le type de valeur renvoyée, s'il s'agit d'une fonction
- Tout cela, encore une fois, vous ne l'inventez pas. Il vous faut tous ces renseignements pour pouvoir travailler, renseignements recueillis dans un bouquin, sur le Net, ou auprès d'un ami si vous avez de mauvaises fréquentations.
- Au total, la déclaration donnera un splendide zorglub du genre :
- `Private Declare Function Duchemol Lib "Coucou32" Alias "Zigopaf" (Byval x As Long, Byval y As Long) As Long`
- Hem. Je sens qu'un petit décodage ne sera pas inutile.
- `Declare` indique qu'il s'agit d'une déclaration API

- Duchemol est le nom que vous choisirez pour votre fonction
- Lib indique qu'il va falloir aller piocher dans une DLL...
- ...DLL dont le nom est Coucou32
- Alias "Zigopaf" est un paramètre éventuel, souvent omis par VB car le nom d'alias est le même que le nom de la DLL
- X et Y sont les paramètres envoyés à la fonction, avec leur type
- le "As Long" final indique le type de valeur retournée par la fonction.
- Ouf.
- Une fois cette fonction d'appel API déclarée, vous pourrez vous en servir comme de n'importe quelle fonction, avec par exemple en cours de programme :
- Toto = Duchemol(45, 156)
- Et voilà. Finalement, c'est simple, non ?

## Les contrôles

- Heureusement, ce qui sauve le programmeur VB (sans parfois qu'il le sache), c'est qu'à chaque fois qu'on appelle des DLL, on n'est pas systématiquement obligé de passer par l'API. Il existe une autre voie, bien plus facile. Cette voie, ce sont les... contrôles !
- Mais oui ! Les contrôles ne sont que des programmes qui servent d'intermédiaire entre le développeur et les DLL de Windows, intermédiaire qui se charge de traduire la discussion en la rendant nettement plus accessible.
- Quand on y réfléchit un peu, tout ça se tient parfaitement. Depuis le début de ce cours, qu'avons nous fait ? Nous avons employé des boutons, des cases à cocher, des listes, etc. autrement dit des éléments de Windows>. Cela signifie que les boutons, les cases à cocher, et tutti quanti existent dans Windows sous forme de DLL. Ce sont ces DLL que nos logiciels classiques (Word, Excel ou Photoshop) utilisent en permanence. Et ce sont ces mêmes DLL que nous avons pu utiliser nous aussi pour écrire nos propres applications.
- Simplement, nous n'avons pas eu à programmer d'appels API lorsque nous avons besoin de tout cela (on s'en serait rendu compte !). Pourquoi ? Parce que d'autres programmeurs, en l'occurrence les auteurs du langage Visual Basic, ont écrit avant nous de petits programmes nous présentant ces DLL sous forme d'objets réutilisables, avec leurs propriétés et leurs méthodes.

- Dès lors, cela signifie que si l'on a besoin d'un élément de Windows, il faut toujours chercher à passer par un contrôle, et ne se résoudre à passer par l'API qu'en dernier recours, si on n'a trouvé aucun contrôle qui effectue la tâche voulue.
- On peut classer les contrôles VB en quatre catégories :
- un certain nombre de contrôles sont fournis en standard avec VB. Ce sont ceux que nous avons explorés tout au long de ce cours. Même si nous en avons laissé un ou deux de côté, ces contrôles ne devraient à l'heure actuelle plus avoir guère de secrets pour vous...
- d'autres contrôles se tiennent en réserve dans VB, prêts à être exploités (par exemple, pour gérer des fonctions multimédia). Pour les utiliser, il faut aller les charger par le menu Projet – Composants. Malheureusement, l'aide disponible sur ces contrôles n'est pas toujours très prolix, pour dire le moins... Concrètement, on ne peut utiliser ces contrôles que muni d'une documentation qui n'existe que dans certains ouvrages spécialisés.
- il existe une autre possibilité : des programmeurs, qui mettent des contrôles supplémentaires à votre disposition sur le Net, à titre plus ou moins gracieux. Ces contrôles sont souvent très utiles, et sont un excellent moyen pour ne pas avoir à réinventer quelque chose qui l'a déjà été. Ils se présentent toujours sous la forme de fichiers \*.OCX.

- Il suffit de les télécharger - si possible dans le bon répertoire, celui qui contient déjà tous les autres fichiers \*.ocx -, et de les activer dans VB via la même manœuvre que précédemment. Dans ce cas, la documentation est de qualité variable. Au pire, le contrôle est livré seul, sans explication, ce qui promet généralement de longues heures de tâtonnement pour comprendre le fonctionnement de ses propriétés. Au mieux (et ce n'est pas rare), le développeur vous fournit un fichier d'aide détaillant propriétés et méthodes, ainsi qu'un exemple d'utilisation. Dans ce cas, la vie est merveilleusement belle.
- enfin, un programmeur averti sera capable de créer lui-même des contrôles. Mais cela sort quelque peu des modestes ambitions de notre propos. Sachez toutefois que cela n'a rien d'insurmontable.
- A signaler que si votre application emploie des contrôles non standard, vous devez veiller, tout comme pour des fichiers images, à ce que l'installateur repère ces contrôles et les écrive dans le bon répertoire des machines sur lesquelles votre application sera distribuée.
- Je termine cette partie par un exemple qui en vaut un autre, celui du contrôle qui gère les DLL des boîtes de dialogue communes (Fichier-Ouvrir, Fichier-Imprimer, etc.) : le contrôle Comdlg32.ocx

- Celui-ci permet d'utiliser la DLL correspondante, dont on a parlé plus haut (comdlg32.dll). Grâce à ce contrôle, on disposera donc des menus Fichier - Ouvrir, Fichier - Enregistrer, etc., sans - presque - avoir à rien faire.
- Avant toute chose, il faut bien sûr commencer par charger le contrôle comdlg32.ocx dans VB, car ce contrôle, par défaut, ne vous est pas proposé. Rappel : pour ce faire, il faut passer par la commande Projet – Composants, et le choisir dans la liste.
- Ensuite, l'astuce, c'est qu'avec ce seul contrôle, on peut ouvrir six boîtes de dialogues différentes. Pour ce faire, il suffit d'appliquer à ce contrôle la méthode appropriée :

| <b>Boîte de dialogue</b> | <b>Méthode à utiliser</b> |
|--------------------------|---------------------------|
| Fichier - Ouvrir         | ShowOpen                  |
| Fichier - Enregistrer    | ShowSave                  |
| Couleur                  | ShowColor                 |
| Police                   | ShowFont                  |
| Imprimer                 | ShowPrinter               |
| Aide                     | ShowHelp                  |



- Avant de lancer telle ou telle méthode pour ouvrir la boîte de dialogue correspondante, il conviendra d'ajuster les propriétés du contrôle, afin de préciser éventuellement des filtres (pour Fichier - Ouvrir), un titre pour votre boîte, etc. De même, la réponse de l'utilisateur sera stockée, comme d'habitude, dans une autre propriété du contrôle. On a ainsi, entre autres, pour la boîte de dialogue Fichier - Ouvrir :
- FileName : qui récupère le nom complet (avec le chemin) du fichier sélectionné par l'utilisateur
- FileTitle : qui récupère le nom du fichier sélectionné par l'utilisateur sans le chemin
- Filter : qui définit le(s) type(s) de fichiers proposés par votre boîte de dialogue
- FilterIndex : qui définit lequel des filtres doit être utilisé
- InitDir : qui fixe le répertoire initial
- DialogTitle : qui donne titre de la boîte de dialogue etc.

- Pour les propriétés des autres boîtes de dialogue communes, cf. les bouquins ou le Net !
- L'affichage d'un Gif animé, la lecture d'un fichier mp3 et la gestion des liens hypertexte sont des tâches qui ne sont assumées par aucun contrôle standard de Visual Basic. Mais pour chacune d'elle, des contrôles gratuits existent... quelque part ! Effectuez la recherche sur le Net, trouvez des contrôles adéquats et programmez-les. Vous ne trouverez pas forcément les mêmes que ceux dont les corrigés donnent la référence ; peu importe, seul le résultat compte !  
Afin de pouvoir lancer les exécutables, un fichier est à chaque fois nécessaire, fichier qui devra être téléchargé dans le même répertoire que l'exécutable.

## Partie 12

- Initiation à la P.O.O.
- Et si je vous disais que toute ce qu'on a fait jusque là n'était qu'une longue introduction ? Que les véritables délices de la programmation objet vous restent encore inconnus ? Que seules les lignes qui suivent pourront vous emporter vers les cimes de la béatitude informatique ? Et que j'ai dû forcer sur la bouteille hier soir pour écrire des choses pareilles ce matin ? Le croiriez-vous ?
- En effet, jusqu'à présent, nous n'avons fait qu'apprendre à utiliser des éléments préfabriqués - par d'autres - pour construire nos applications : ces éléments sont les fameux contrôles, que je ne vous présente plus. Cela nous a permis, du point de vue de l'interface, de franchir un pas considérable par rapport à la programmation classique. Ainsi, nous avons découvert la programmation événementielle. C'est déjà bien, me direz-vous.

- Certes, vous répondrai-je, car aujourd'hui, j'ai décidé de ne pas être contrariant. Mais réfléchissons un peu plus loin : lorsque nous avons eu des informations à mémoriser ou à traiter, nous avons utilisé les bons vieux outils de la programmation classique : des variables, voire des tableaux ou des variables structurées. Et au pire du pire, des tableaux de variables structurées.
- Or, rappelez-vous ce que nous avons vu au tout début de ce cours, lorsque nous avons défini ce que sont les objets : les objets sont, avant toute autre chose, une manière nouvelle de structurer les informations (j'avais employé le terme de "super-variables"). De tout cela, qu'avons nous fait ? En réalité, rien : nous l'avons mis de côté, pour ne nous servir que d'une catégorie particulière d'objets, les contrôles. A peine avons-nous effleuré, pour un besoin précis ou pour un autre, des objets comme App. Autrement dit, même si c'est dur à entendre, il est de mon devoir de vous parler franchement : jusqu'à présent, nous n'avons pas réellement abordé la programmation objet.

- Alors, avouez-le, ce serait vraiment trop dommage de se quitter sans remédier à cette grave lacune. Car si les fainéants nous disent que l'humanité a bien vécu quelques millions d'années sans connaître la programmation objet et qu'elle ne s'en est pas portée plus mal, nous leur répondrons sans hésiter qu'avec des raisonnements pareils, on en serait encore à tailler des silex.
- Résumons-nous, en regroupant dans un tableau les concepts et le vocabulaire lié à la manipulation des objets et des variables. Rassurez-vous, on reviendra sur tout cela dans pas longtemps

|                                          | <b>Variable</b>                     | <b>Objet</b>                                                       |
|------------------------------------------|-------------------------------------|--------------------------------------------------------------------|
| <b>contient</b>                          | <b>des informations</b>             | <b>des informations (les propriétés) et du code (les méthodes)</b> |
| <b>est désigné par un...</b>             | <b>nom de variable</b>              | <b>nom d'objet</b>                                                 |
| <b>le "moule" s'appelle</b>              | <b>un type ou<br/>une structure</b> | <b>une classe</b>                                                  |
| <b>On peut fabriquer ce "moule" dans</b> | <b>un module</b>                    | <b>un module de classe</b>                                         |

- **Mener une analyse objet**
- Si l'on veut développer une application en raisonnant réellement en termes d'objets, c'est dès le départ, bien avant d'écrire le code, qu'il faut orienter l'analyse dans cette direction.
- Mener correctement l'analyse d'un projet complexe en termes objets suppose un savoir-faire extrêmement long à acquérir. Les détracteurs de la méthode objet parlent d'ailleurs volontiers à ce propos d'usine à gaz. Et si l'on jette un oeil sur les ouvrages traitant de la question, on est généralement tenté de leur donner raison, tant le vocabulaire est imperméable à tout être humain normal, et tant les explications fournies fourmillent d'abstractions qui les rendent parfaitement obscures.
- Cependant, pour se faire une idée de la chose, on peut aborder la question d'une manière simple, en menant une rapide analyse d'un problème pas trop compliqué. Encore une fois, j'insiste, les lignes qui suivent se veulent être une très brève illustration, et nullement un exposé couvrant tous les recoins de la chose.

- Imaginons donc que nous voulions programmer un jeu de Scrabble pour deux joueurs.
- Analyse classique
- Dans une analyse classique, on ferait l'inventaire des données qu'il faudrait coder, et de la manière dont ces codages seraient effectués au mieux.
  
- **Plateau de jeu** : pour chaque emplacement du plateau de jeu, on doit savoir s'il s'agit d'une case simple, ou d'une case avec bonus (mot compte double, lettre compte triple, etc.). On doit également savoir si une lettre y a été posée, et quelle est cette lettre. Une solution simple consiste à créer un tableau de caractères à deux dimensions. On représenterait les cases vides par un chiffre, correspondant à leur valeur : 0 pour une case simple, 1 pour une lettre compte double, 2 pour une lettre compte triple, etc. Lorsqu'une lettre serait posée, on remplacerait dans la case du tableau ce code par la lettre en question.
- **Valeur des lettres** : un autre aspect à gérer est de mémoriser les valeurs des différentes lettres : au Scrabble, le A vaut 1 point, le B vaut 3 points, etc. Il y a plusieurs moyens de répondre à ce problème. Un d'entre eux consiste à remplir un tableau de 26 nombres, en y mettant, dans l'ordre, les valeurs des lettres. Il suffira ensuite de retrouver que le M est la 13e lettre de l'alphabet, pour aller chercher la treizième valeur de ce tableau.

- **Nombre de lettres** : toutes les lettres ne sont pas disponibles en quantités égales dans le jeu. Fort heureusement, il y a davantage de E que de W, par exemple. Là aussi, il faudra stocker et mettre à jour cette information, par exemple en tenant à jour un tableau de 26 numériques indiquant les disponibilités restantes pour chaque lettre.
- Ces quelques lignes ne représentent bien entendu pas une véritable analyse : elles ne font qu'effleurer l'affaire, mais je les ai écrites pour montrer (en fait, normalement, ce n'est qu'un rappel) quelles questions l'on se pose quand on programme de manière traditionnelle.



- Analyse Objet

- Passons maintenant à un bref aperçu de la manière dont un programmeur objet se poserait les problèmes.

- Au scrabble, que fait-on ? On pioche des jetons, qu'on met sur les réglettes des joueurs, puis sur le plateau. Nous devons donc programmer les tribulations de différents objets de type jeton.

- Le plateau est formé de cases, qui ont différentes caractéristiques (leur emplacement, leur valeur, le fait qu'elles soient vides ou occupées par des jetons...). Les cases du plateau seront elles aussi des objets maniés dans le jeu Scrabble.

- Dès maintenant, remarquons que ce qui entre en compte n'est pas, pour l'essentiel, l'aspect graphique. Certes, on voit les jetons et les cases (encore que, pas toujours, il y a des jetons cachés). Mais ce n'est pas le problème, tout au moins, pas pour l'instant. Ce qui compte, pour parler comme les vrais informaticiens, c'est l'aspect fonctionnel des choses. Ce qui compte, ce sont les "choses" qu'on manipule (j'emploie un mot fourre-tout exprès, car ces "choses" peuvent être tout et n'importe quoi). Mais en l'occurrence, comme souvent, le mieux, c'est de faire simple et direct. Nos objets sont donc des jetons et des cases.

- Continuons, en examinant ces objets de plus près.

- **Les cases** : chaque case se caractérise par sa position sur le plateau (numéro de ligne, numéro de colonne), par son type de bonus, et par son contenu (le jeton qui vient éventuellement l'occuper). Accessoirement, la case se caractérise aussi par un graphisme différent (les "mots compte double" n'ont pas la même tête que les "lettre compte triple").
- **Les jetons** : chaque jeton du jeu possède une lettre faciale, une valeur, et un emplacement (est-il dans le sac, sur la réglette de tel joueur, ou sur telle case du plateau).
- Là aussi, ces quelques lignes sont bien sommaires, et ne font qu'indiquer la direction dans laquelle une analyse complète devrait être menée.
- Ce qu'on peut établir dès à présent, c'est que notre jeu manipule deux types d'objets : : le type "cases", et le type "jetons". Toutefois, en bons programmeurs, nous ne parlerons pas de "types" d'objets (ce serait sans doute trop clair) mais de classes d'objets. Nous aurons donc deux classes : la classe cases, et la classe jeton. Exactement comme nous avons jusqu'à maintenant manié des objets de la classe Bouton de Commande, de la classe Case à Cocher, etc.

- Chaque case du plateau, chaque jeton du jeu est donc un représentant de la classe Cases ou de la classe Jeton. Mais là encore, attention Léon : pas question de parler de "représentant", ce serait d'une vulgarité affligeante. L'usage impose le terme fleuri d'instance de classe. Nous dirons donc que les cases individuelles sont des instances de la classe Cases, et que chaque jeton est une instance de la classe Jeton. De la même façon, jusqu'à présent, chaque bouton de commande que nous avons créé était une instance de la classe Boutons de Commande. Ainsi, dans la vie courante, vous saurez dorénavant que chaque gâteau est une instance de la classe Moule à Gâteau. Pensez-y la prochaine fois que vous commanderez quelque chose dans une boulangerie. Effet 100% garanti.
- Enfin, chaque caractéristique d'un objet individuel (pour les cases, le numéro de ligne, le numéro de colonne, le type du bonus, etc. ou pour les jetons, la lettre représentée, le nombre de points, l'emplacement dans le jeu) est une propriété de l'objet en question. Mais ce gros mot-là, vous le connaissiez déjà.
- Il va de soi qu'en plus des propriétés, nous pourrions éventuellement concevoir des méthodes pour nos objets, c'est-à-dire des traitements en quelque sorte préfabriqués sur leurs propriétés.

- Quoique dans l'exemple du Scrabble, aucune ne s'impose immédiatement à l'esprit ; on pourrait toujours se forcer pour en inventer, mais franchement, on n'est pas là pour imaginer des difficultés là où il n'y en a pas.
- Enfin, une analyse qui serait poussée plus loin devrait également recenser les événements que chaque objet de nos nouvelles classes pourrait recevoir, et les réactions de l'application à ces événements. Là encore, je ne fais que signaler l'existence éventuelle de ce problème, car dans notre exemple, les événements classiques (clic, drag and drop, etc.) devraient largement suffire.
- Parvenus à ce stade, on voit qu'en fait, on a recensé peu ou prou les mêmes choses que dans l'analyse classique. Alors pourquoi tout ce barouf ? Parce que nous avons organisé nos idées différemment, et que c'est cette organisation différente qui va nous simplifier la vie au niveau de la programmation (du moins, si l'analyse a été bien menée, que nos classes et nos propriétés sont pertinentes).
- La programmation objet ne dispense d'aucune abstraction mentale nécessaire à la programmation traditionnelle. En ce sens, elle ne constitue aucun progrès. Mais elle rend l'écriture de l'application beaucoup plus proche des événements tels qu'ils se déroulent dans la réalité. Et en cela, elle permet de développer un code beaucoup plus lisible. Le travail en équipe des programmeurs sera donc plus facile à coordonner, et la structure du programme sera plus légère et plus claire. Enfin, si c'est réussi.

- Un des paris de la programmation objet, c'est d'alourdir un peu la phase d'analyse, pour gagner beaucoup sur la phase de développement.
- **Objets et Visual Basic**
- Mais, si ce n'est que cela, direz-vous, ne serait-ce pas tout bêtement donner un nouveau nom à une vieille marmite ? Car après tout, construire des variables par agglomération de types simples, il y a très longtemps qu'on sait le faire ! Depuis les langages traditionnels et les bonnes vieilles données structurées ! Les objets ne seraient-ils donc pas tout bêtement des données structurées rebaptisées autrement pour être mieux vendues ?
- Eh bien non, comme on l'a déjà dit, les objets ne sont pas que cela. Ils sont certes des données structurées. Mais ils sont des données structurées avec deux possibilités supplémentaires, ce qui change complètement la donne.
- les objets ne sont pas que des propriétés ; il sont aussi des méthodes. Cela veut dire que si dans un langage traditionnel, on sépare les données et les traitements sur ces données, en programmation objet, à l'inverse, on fabrique des bidules (les objets) qui regroupent les données (les propriétés) et les actions sur ces données (les méthodes). Visual Basic, dans la mesure où il fournit le moyen de construire de tels objets, est un langage objet.

1. les objets possèdent une souplesse remarquable, qui se traduit notamment par la notion d'héritage. Je crée une classe "mère", puis des classes "filles" qui héritent de toutes les caractéristiques de la classe mère, plus d'autres, puis des classes petites-filles, qui héritent des caractéristiques de la classe fille, plus d'autres, etc. Ainsi, par exemple, je crée la classe "animal", puis les classes filles "reptile", "mammifère", "poisson", etc. qui possèdent toutes les propriétés et méthodes des animaux, plus d'autres spécifiques. Et ensuite, je crée les classes "félin", "canin", "rongeur", etc, qui héritent les propriétés et méthodes des mammifères, en ajoutant des propriétés et méthodes propres et ainsi de suite. Ca ne paraît pas comme ça, mais c'est une manière de procéder très souple et très puissante pour modéliser des données complexes.

Or, Visual Basic ne donnant pas le moyen de gérer l'héritage, ce n'est pas un vrai langage objet.

2. D'où le titre ambigu du titre du premier chapitre de ce cours : "VB, un langage (presque) objet". La boucle est ainsi bouclée, tout est dans tout et réciproquement.

- **Le code de la programmation objet**

- Comment, une fois l'analyse faite, traduire tout cela en Visual Basic ? C'est ce qu'on va voir maintenant.
- 2.1 Créer une nouvelle classe
- Pour disposer dans notre application des objets de classe Jeton et Cases, il nous faut bien évidemment commencer par créer ces deux classes, qui n'existent pas a priori dans Visual Basic (à la différence des classes de contrôles, qui sont là toutes prêtes à être utilisées).
- Une classe se définit par un certain nombre de lignes de codes écrites dans un emplacement spécifique appelé Module de Classe.
- Donc, de même que nous connaissons les Form et les Modules, nous avons à présent affaire à un troisième type de bidule dans lequel taper du code. On définit une classe et une seule par module, ce qui implique de manière imparable qu'on devra avoir autant de Modules de classes que de classes d'objet. En l'occurrence, pour notre mini-scrabble, il nous faudrait créer deux modules de classes : un pour les jetons, un pour les cases du plateau.
- De plus, le nom de la classe sera celui du module. Pour changer le nom de cette classe (de ce module) il suffit d'aller voir les propriétés de notre module de classe, comme nous changions jusque là par exemple le nom d'un contrôle posé sur une feuille.
- Dans notre exemple, nous devons donc créer un module Jeton et un module Case.

## Définir les propriétés de la classe

- Le principe de base est le suivant :
- à chaque propriété d'une classe doit correspondre une variable du module de classe. C'est cette variable qui stockera la valeur de la propriété pour chaque objet de la classe.
- la lecture de la propriété (obligatoire) sera gérée par une procédure de type Property Get
- l'écriture de la propriété (facultative) doit être gérée par une procédure de type Property Let

En théorie, nous devons nous demander pour chaque propriété si elle devra fonctionner en lecture - écriture, ou en lecture seulement (quitte à asséner une banalité, je rappelle qu'une propriété ne peut pas exister en écriture sans exister en lecture). Autrement dit, est-ce que je donne à mon propre programme le droit de modifier telle ou telle propriété ? Cette question n'est pas d'un grand intérêt pour une petite application autonome, mais elle peut devenir plus épineuse dans le cas d'une application plus grosse, où il serait dommage qu'un bout de code malheureux casse tout sur son passage. Cela dit, 99 fois sur 100, on rend systématiquement les propriétés accessibles tant en écriture qu'en lecture. Donc, sauf raison impérieuse, chaque propriété sera l'occasion d'écrire deux procédures Property.



- Donc, je me répète, mais pour chaque propriété, la procédure Property Get, correspondant à la lecture, est obligatoire, la procédure Property Let, correspondant à l'écriture, est facultative (mais très souvent présente quand même !).
- Définissons par exemple la propriété "valeur" d'un jeton, en admettant que nous souhaitons que cette propriété soit accessible tant en lecture qu'en écriture. Nous écrivons dans notre module de classe Jeton :
- Private Tutu as Integer

```
Public Property Let Valeur (ByVal Nb As Integer)
 Tutu = Nb
End Property
```

```
Public Property Get Valeur() As Integer
 Valeur = Tutu
End Property
```

- Fouillouillouille... Essayons de comprendre tout ce charabia.

- Pour cela, le mieux est d'imaginer ce qui va se passer lorsqu'on va utiliser un des objets Jeton. Quelque part dans le code principal, on va créer les jetons les uns après les autres (on verra comment faire cela dans un moment). Et puis, pour chaque jeton créé, on va affecter ses propriétés. Supposons que nous en soyons à créer un des jetons "B" du jeu, qui au Scrabble, valent trois points. Si le nom de ce jeton est Caramel(i), car nous définirons vraisemblablement tous les jetons du jeu comme un groupe, nous aurons une ligne qui ressemblera à :
  - Caramel(i).Valeur = 3
  - Regardons ce qui va se passer lorsque le programme exécutera cette ligne.
  - La propriété Valeur étant utilisée en écriture, la procédure Property Let Valeur sera immédiatement exécutée. Celle-ci devant transmettre une information à l'objet (ici, le nombre de points), elle comporte obligatoirement un paramètre en entrée (que j'ai appelé Nb). Ce paramètre Nb, dans mon exemple, vaut 3. La ligne de code suivante place la valeur de Nb dans la variable Tutu, qui vaut donc à présent elle aussi 3. Et le tour est joué : Tutu, variable privée de la classe Jetons, a pour rôle de stocker la propriété Valeur de mes objets de type Jeton.

- Dans l'autre sens, ça marche tout aussi bien : si j'utilise au cours de mon programme ma propriété Valeur en lecture, comme par exemple en faisant :
- `Points = Points + Caramel(i).Valeur`
- Ce code, qui appelle la propriété Valeur, déclenche illico la procédure Property Get Valeur. Celle-ci va en réalité marcher comme une fonction : elle va consulter combien vaut la variable Tutu, et renvoyer dans Valeur le contenu de cette variable.
- Soit dit en passant, vous devez comprendre que ce mode de fonctionnement nous permet d'effectuer, si nous le souhaitons, un contrôle élaboré sur les propriétés de nos objets. Par exemple, les valeurs des lettres, au Scrabble, sont exclusivement 1, 3, 8 et 10. Afin d'être sûr et certain que la propriété Valeur d'un jeton ne pourra jamais être autre chose que cela, je pourrais transformer ma procédure Let en :
- ```
Public Property Let Valeur (ByVal Nb As Integer)
If Nb = 1 or Nb = 3 or Nb = 8 or Nb = 10 Then
    Tutu = Nb
EndIf
```

- Et toc ! Toute tentative d'affecter à la Valeur d'un Jeton un autre nombre que 1, 3, 8, ou 10 se solderait par un échec cuisant. On peut ainsi blinder ses objets pour pas cher, et s'assurer qu'ils se comporteront toujours très exactement comme on le voulait.
- Pour finir, une excellente nouvelle : c'est qu'il existe dans VB un petit outil qui vous évitera de frapper vous-mêmes ce code fastidieux. Il suffit pour cela d'aller dans le menu Outils - Gestionnaire de compléments, et de cocher la case "VB Class Builder Utility". Cela rendra ensuite disponible, dans ce même menu Outils, la commande Générateur de Classes.
- Grâce à celle-ci, vous pouvez définir en deux clics le nom et le type de chacune des propriétés souhaitées pour une classe de votre cru, et VB se chargera lui-même d'écrire toutes les procédures correspondantes. Merci, monsieur VB.

- Définir les méthodes de la classe

- Définir les propriétés, c'est bien. Mais définir les méthodes, c'est encore mieux, et, coup de chance, ce n'est pas difficile à faire. En fait, à chaque méthode doit correspondre une procédure qui modifie certaines des propriétés de l'objet

- ▬

- Nous voulons par exemple créer la méthode Tirer, qui consiste à sortir un jeton du sac pour l'attribuer à tel ou tel joueur. On suppose qu'on a créé la propriété Emplacement des jetons, propriété numérique qui désigne le numéro du joueur possédant actuellement le jeton (comme au Scrabble il n'y a que quatre joueurs au maximum, on pourrait alors supposer que la valeur 5 désigne un jeton encore dans le sac et la valeur 6 un jeton posé sur le plateau de jeu).

- Pour fonctionner, la méthode Tirer aura besoin d'un argument, qui sera le numéro du joueur qui vient de tirer ledit jeton.

- Au total, le code donnera :
- ```
Public Sub Tirer (Toto As Object, Nb As Byte)
Toto.Emplacement = Nb
End Property
```
- On remarque qu'un argument obligatoire d'une procédure de méthode est l'objet auquel s'applique la méthode. Et si, comme c'est le cas ici, la méthode exige de surcroît des arguments, ces arguments se traduisent par des paramètres supplémentaires de la procédure définissant la méthode.
- Toujours est-il qu'une fois ceci fait, dans le corps même de l'application, le tirage du caramel numéro i par le joueur numéro j pourra du coup s'écrire :
- ```
Caramel(i).Tirer (j)
```
- Et voilà.

- Créer des instances de classe
- Il ne nous reste plus qu'à voir comment le programme peut lui-même générer de nouveaux objets (ou en supprimer) en cours de route. En fait, ce n'est là que l'extension aux objets "faits main". d'un procédé déjà largement abordé à propos de objets préfabriqués que sont les contrôles (voir [partie 10](#)).
- De même que la création d'une variable s'effectue grâce aux mots-clés Dim et As, la création d'un nouvel objet va elle aussi emprunter les mêmes chemins :
- Dim Toto As New Jeton
- Ici, Toto est le nom (propriété Name) de l'objet, et Jeton correspond obligatoirement au nom d'un module de classe (ou alors, il s'agit d'une classe de contrôles, comme Form, CommandButton, etc.)
- Si l'on voulait créer dynamiquement les 95 jetons du jeu, le plus simple serait sans doute de dire :

- For i = 0 to 94
Dim Caramel(i) As New Jeton
Next i
- Et le tour serait joué.
- Quant à la destruction d'un objet, elle est d'une simplicité biblique
⚡
- Set Toto = Nothing
- Et l'objet disparaît pour toujours dans le néant intersidéral.
Comme disent les comiques, "poussière, tu retourneras
poussière..."
- Avant d'en terminer, j'ajouterais juste qu'il est également possible
de stipuler les événements auxquels devront réagir les objets de
telle ou telle classe. Mais là, on commence à aller un peu loin pour
les modestes objectifs de ce cours. On s'en tiendra donc sagement
là... pour le moment en tout cas.

- Ce cours de Visual Basic est maintenant terminé. Mais si
vous avez aimé cela et que vous comptez vous servir de ce
que vous avez appris, vous aurez compris depuis longtemps
que vos ennuis, eux, ne font que commencer...

