Programmation Visual basic

Cours

Rédaction
Jean-Yves Février

Coordination pédagogique
Christine Guillebault

Direction pédagogique
Bruno Viale

Directeur de publication
Jean-Luc Faure

Impression
Cned
Institut de Poitiers-Futuroscope

Séquence 1

Introduction

1. Objet de ce cours et pré-requis

Si vous lisez ces lignes, c'est que vous vous êtes inscrit auprès du CNED pour apprendre à programmer dans le langage Visual Basic (VB).

La programmation est une discipline informatique permettant d'écrire des programmes qui s'exécuteront sur des ordinateurs. Ces programmes ont pour objet l'automatisation des tâches lourdes et répétitives qui, sinon, seraient faites à la main. Cela va de la facturation à la gestion des stocks en passant par le suivi d'un cheptel ou autre. Je pourrais même parler de jeux et de sites internet, mais ces thèmes sont hors sujet : nous nous limiterons à l'informatique orientée vers des thèmes de gestion.

Il y a deux niveaux de maîtrise de la programmation :

- les bases sont constituées de l'apprentissage de l'algorithmique. Cela vous permet de comprendre la logique des programmes et de pouvoir les écrire de façon théorique ;
- il faut ensuite passer de l'algorithmique à la programmation concrète. Cette étape est beaucoup plus simple que la précédente car elle consiste uniquement en une phase de traduction de votre algorithme dans un langage de programmation réel, tel le C, VB, Delphi…

Le cours que vous avez entre les mains correspond au second niveau. Vous devez donc connaître l'algorithmique pour en tirer partie.

Attention, l'objet du cours n'est pas de faire de vous des développeurs professionnels. Pour cela, il faudrait suivre une formation d'informatique. Mon objectif est ici de former des gens motivés qui souhaitent apprendre à écrire correctement des programmes pour leur propre compte.

2. Contenu exact de ce cours

J'ai dit que le passage de l'algorithmique vers un langage était très simple. Ce n'est qu'à moitié vrai.

En effet, l'algorithmique a pour objet la découverte des instructions (le *si*, le *pour*…). Comme ces instructions sont basées sur les vraies (celles des différents langages), il me suffit de vous dire que le *si alors sinon* algorithmique s'écrit *if then else* sous VB et, ma foi, une bonne partie du travail est terminé. Le reste, ce serait la découverte des quelques subtilités qui distinguent un langage d'un autre et qui font qu'en choisissant VB, vous programmez en VB et non pas en Delphi. Mais bon, comme ces langages restent similaires (ils sont tous deux procéduraux), les différences sont avant tout syntaxiques.

Il y a dix ans, le cours se serait donc vite arrêté.

Mais depuis dix ans, les langages de programmation ont nettement évolué. Les langages Basic, C et Pascal de l'époque sont devenus Visual Basic, Visual C et Delphi (Visual Pascal). Qu'apporte ce nouveau concept *visual* ? Le support de l'interface graphique proposée par les systèmes d'exploitation modernes[1].

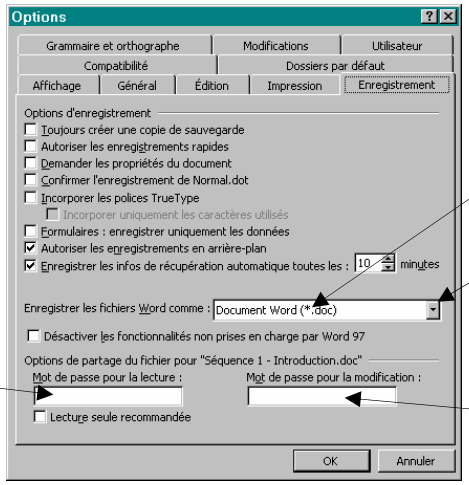
Nos langages de programmation visuels possèdent donc deux composantes :

[1] Par exemple l'interface Windows, apparue avec Windows 95 sorti en 1995. Attention, Windows n'est pas le seul, il existe aussi les systèmes Unix également dotés d'une interface graphique et d'autres systèmes encore.

- d'une part, la partie purement algorithmique, soit les différentes instructions permettant de transcrire l'algorithme dans le langage ;
- d'autre part, tout ce qu'il faut pour manipuler l'interface graphique Windows (ou une autre, mais ici, c'est VB que nous étudions et il n'existe que sous Windows).

Notez bien que l'interface graphique est issue du système d'exploitation et pas du langage. Prenons le concept de zone de texte, qui est l'un des contrôles proposés par Windows[1]. Toutes les applications tournant sous Windows disposent de ce contrôle dont l'objet est la saisie d'un texte.

Voici une copie écran d'une fenêtre de Word ; elle dispose de deux zones de texte.



Attention, ceci n'est pas une zone de texte, mais une zone de liste modifiable. On le voit à la présence de cette flèche là.
(Nous étudierons ensemble les différents contrôles, aucun pré-requis n'est nécessaire sur ce point.)

Une zone de texte ici

Une autre zone de texte là

L'objet des langages de programmation est d'écrire des programmes. Il est évident que l'interface du programme (ce que l'on voit à l'écran) doit être conforme aux standards du moment. Un développement réalisé maintenant sans exploiter l'interface graphique qui apporte un confort formidable à l'utilisateur, cela n'a aucun sens.

Les langages de programmation se doivent donc de permettre au développeur (à savoir vous) d'exploiter les différents constituants de l'interface graphique du système d'exploitation ; cela passe de la souris aux fenêtres en passant par les contrôles.

Comme ces constituants viennent du système d'exploitation et pas du langage, on les retrouvera à l'identique dans tous les langages. Une zone de texte reste une zone de texte, que ce soit avec Delphi, VB, Visual C, Visual J++…

Cela dit, la façon de manipuler ces différents constituants dépendra de la syntaxe et de la philosophie du langage de programmation utilisé. Et c'est cela que nous étudierons avec VB.

Pour bien comprendre, je vous propose la métaphore suivante : toutes les voitures proposent le concept de boîte de vitesses, mais sa mise en œuvre dépend du constructeur et du modèle : la boîte peut être manuelle, automatique, sous la forme de boutons sur le volant…

Finalement, ce cours va aborder quatre choses :

- la traduction des instructions algorithmiques sous VB ;
- les concepts algorithmiques poussés (fichiers par exemple) que nous aborderons directement sous VB, sans passer par la phase algorithmique qui n'apporterait pas grand chose ;
- les spécificités de VB ;
- la façon de manipuler l'interface graphique sous VB. J'en profiterai pour vous présenter les différents constituants de cette interface ;
- la présentation du meilleur ami du développeur, le débuggeur.

[1] Ne vous inquiétez pas si ces concepts de zone de texte et de contrôles ne vous parlent pas. Nous les étudierons en détail dans la suite de ce cours.

3. Présentation du support de cours

Ce cours a été conçu pour pallier au maximum les difficultés de l'apprentissage à distance : les notions à retenir (définitions…) sont mises en avant et des exercices et questions sont présents tout au long du support pour vous permettre de vérifier votre compréhension.

Mais j'insiste sur le point suivant : quelle que soit la qualité pédagogique de ce cours, il ne vous permettra pas d'assimiler la programmation par simple imprégnation visuelle. Vous devrez fournir un travail d'apprentissage (le cours), de réflexion (toujours le cours) et d'entraînement (les exercices).

Le cours est constitué de deux fascicules : un contenant le cours proprement dit et un contenant la correction de tous les exercices et des TD.

3 A. Organisation

Le fascicule de cours contient différentes choses :

- neuf séquences de cours correspondant aux différents savoirs à acquérir ;
- quarante-deux exercices intégrés aux séquences de cours. Vous devez faire ces exercices quand vous arrivez dessus puis aller consulter la correction. Attention, ces exercices permettent de vérifier votre assimilation du cours. Leur corrigé, très détaillé, ne doit pas être négligé : j'y présente des situations, des techniques, des idées et des raisonnements qui ne sont pas anecdotiques et font **partie intégrante** du cours. S'ils n'y sont pas physiquement, c'est uniquement pour conserver une certaine lisibilité au document ;
- trois séries d'exercices jouant le rôle de travaux dirigés. Elles sont placées à des endroits stratégiques du cours ; vous devez évidemment les faire au moment où vous arrivez dessus.

Les fascicules de correction comprennent :

- la correction des 42 exercices intégrés aux séquences ;
- la correction des séquences de TD.

En plus de vous donner la solution des exercices, j'ai essayé de détailler ces corrections pour envisager les différentes solutions possibles, les erreurs à éviter… Plus que des corrections, ce sont de véritables éléments de cours. Il ne faut donc pas les prendre à la légère !

3 B. Notations

Pour vous aider à identifier les différents constituants du cours, j'ai utilisé les représentations suivantes :

- tout ce qui est précédé d'un caractère cœur (♥) doit être appris par cœur. Cela correspond aux définitions ou explications qu'il est absolument nécessaire de connaître pour s'en sortir en programmation. Quand je dis *apprendre*, ce n'est pas retenir pour l'heure qui suit afin d'épater les convives au prochain repas. Il s'agit d'une vraie leçon, dont vous devez vous souvenir tout au long de votre vie d'informaticien ;
- les exercices intégrés dans les séquences et destinés à vérifier votre compréhension sont numérotés et encadrés par un crayon (comme : ✎Exercice 1✎). Il faut donc les faire au fur et à mesure de la lecture du cours. Leur correction se trouve dans le fascicule *Correction des exercices*.

4. Quelques conseils

Le seul conseil utile que je puisse vous donner est de garder à l'esprit la fable de La Fontaine *Le lièvre et la tortue* : il ne sert à rien de travailler comme un fou ; travaillez plutôt dès maintenant quelques heures par semaine ré-gu-li-è-re-ment (j'aurais pu écrire **régulièrement** ou RÉGULIÈREMENT, mon but étant juste d'insister sur le mot).

La difficulté de l'enseignement par correspondance réside dans le fait que, par définition, vous êtes seul face au cours, personne n'est là pour vous guider, insister sur l'essentiel ou établir la progression du cours.

Pour vous aider à suivre un rythme correct, disons que chaque séquence correspond à un travail d'environ 6 à 8 heures.

Attention à l'*environ* ! Vous avez sans doute le souvenir de vos études où, pour obtenir un même résultat, certains travaillaient toute la soirée et d'autres se contentaient d'être présents en cours. Il en est de même ici. Les 7 heures ne sont qu'un ordre de grandeur signifiant juste que 15 minutes, ce n'est pas assez, mais 25 heures, c'est trop.

Retenez qu'il vaut mieux passer 10 heures sur une séquence et la comprendre parfaitement, que faire exactement 420 minutes (7 heures) en passant à côté de l'essentiel.

De plus, le cours contient des dizaines de petits exercices à faire au fur et à mesure. Plus vous passerez du temps dessus (et c'est conseillé), plus vous risquez de dépasser les 7 heures.

Séquence 2

À quoi sert le *programme* VB ?

Nous allons découvrir l'interface du programme VB et tout ce qu'il permet de faire.

Contenu

Les différentes phases dans l'écriture et l'exécution d'un programme
L'interface utilisateur de VB
Les commandes principales

Capacités attendues

Maîtriser les notions clés de source, exécutable et compilation
Savoir manipuler les commandes essentielles de VB
Utiliser l'aide

1. Rappel du rôle d'un programme source

1 A. La phase de compilation

J'ai dit dans la séquence précédente que nous allions apprendre à écrire des programmes qui s'exécuteront sur l'ordinateur.

En fait, c'est faux : il y a programme et programme. Celui que nous écrirons sera traduit par un outil spécifique (le compilateur) lors de la phase dite de compilation. Le résultat de cette compilation sera un programme qui, lui, sera exécuté par l'ordinateur.

Notez que cette phase de compilation est transparente lorsque le programme est écrit correctement. Elle est tellement transparente que, parfois, certains ne la connaissent même pas.

1 B. Représentation de l'information dans l'ordinateur

1 B 1. Les informations

Quel est l'intérêt du compilateur ? Il est immense. Un ordinateur ne manipule que des *0* et des *1* (le terme technique est *bit*, information valant *0* ou *1*). N'oublions pas que tous les composants de l'ordinateur sont alimentés par du courant. La mémoire vive, notamment, qui contient les programmes en cours d'exécution, contient des charges électriques. Dit autrement, toutes les informations contenues dans la mémoire sont codées à l'aide de charges électriques. Cela justifie nos valeurs *0* et *1* : le *0* correspond à l'absence de courant et le *1* à la présence de courant.

Tout cela pour dire que toute information (votre CV tapé sous Word, une fiche de paie réalisée sur un tableur, le programme VB que nous allons lancer…) en mémoire est codée sous la forme de *0* et de *1*[1]. Évidemment, le codage n'est pas arbitraire, il existe des règles très précises permettant de passer d'une information quelconque à son codage sous la forme de *0* et de *1*. Le codage doit évidemment être décodable, puisque les informations doivent être récupérées : le codage en *0* et *1* est la technique propre à l'ordinateur pour stocker les informations, mais l'utilisateur veut toujours les percevoir sous leur forme habituelle (des nombres, du texte…).

Ainsi :

- lorsqu'une donnée entre dans l'ordinateur (saisie au clavier par exemple), elle est codée sous forme de *0* et de *1* ;
- lorsqu'une donnée doit sortir de l'ordinateur (affichage à l'écran), les *0* et *1* qui la représentent dans l'ordinateur sont décodés pour lui rendre sa forme lisible.

Prenons un exemple.

Envisageons un programme élémentaire et, il faut bien le reconnaître, sans intérêt : l'utilisateur saisit un nombre et le programme affiche ce nombre à l'écran.

Supposons que je saisisse la valeur 73. Elle sera codée en mémoire 01001001[2]. Lorsque le programme devra afficher la valeur, il ira chercher les *0* et les *1* en mémoire (soit 01001001) et les décodera pour donner 73.

[1] Je ne souhaite pas vous faire un cours théorique d'architecture des ordinateurs. Je suis donc un peu imprécis, car ce qui m'intéresse ici, ce n'est pas tant la façon dont les informations sont stockées que la conséquence, pour nous, de ce mode de stockage.

[2] Pourquoi précisément cette succession de *0* et de *1* et pas, par exemple 11001100 ? C'est hors sujet ici, l'important, c'est que le codage fonctionne. Pour les curieux, je précise quand même qu'il s'agit d'un

Voici d'autres exemples de codage :

- le mot *Teckel* (je raffole de ces petits animaux) sera codé 01010100011001010110001101101011011001010110110[1] ;
- le nombre réel (« à virgule ») – 823 086 683 711 594 488 011,716 506 419 23 sera codé 10000000111010110000111100000111110001111100011[2].

1 B 2. Les instructions

Un programme informatique possède deux composantes : d'une part, les informations, les données qu'il manipule et, d'autre part, le programme lui-même, à savoir les différentes instructions.

Nous venons d'étudier la façon dont les différentes informations manipulées par l'ordinateur étaient stockées dans sa mémoire. Voyons maintenant comment les instructions sont stockées.

En fait, il n'y a pas de mystère : comme toute chose manipulée par l'ordinateur, les instructions seront stockées sous la forme de *0* et de *1*.

Voyons l'instruction algorithmique suivante :

```
si Nombre > 1
   alors
      Nom := "Teckels"
   sinon
      Nom := "Teckel"
fin-si
```

Prenons un peu d'avance… sous VB, cette instruction s'écrira ainsi :

```
01  if Nombre > 1 then
02        Nom = "Teckels"
03     else
04        Nom = "Teckel"
```

Comment coder cette instruction avec des *0* et des *1* ? Ce n'est pas possible, car cette instruction est complexe. Attention, pas *complexe* dans le sens compliqué, mais *complexe* car constituée de plusieurs éléments. En effet, l'instruction *if* présentée ci-dessus est écrite sur cinq lignes. Cette instruction contient deux instructions :

- une affectation en ligne 2 ;
- une autre affectation en ligne 4.

---

changement de base. Nous comptons en base 10 et manipulons donc les chiffres de *0* à *9*. La base 2 (appelée base binaire) ne possède que les chiffres *0* et *1*. Ainsi, les nombres *0, 1, 2, 3, 4, 5*… en base 10 correspondent respectivement aux nombres *0, 1, 10, 11, 100, 101*… en base 2. Si vous continuez à compter comme cela, vous vous rendrez compte que 73 en base 10, c'est 1001001 en base 2. Je rajoute un *0* devant ce nombre car les bits sont manipulés par paquets de 8. Ce calcul mathématique, pas très facile pour nous, est très simple pour l'ordinateur. Pour retrouver la valeur initiale, il suffit de faire l'opération inverse pour passer de la base 2 à la base 10.

[1] Comment ai-je obtenu cette suite de *0* et de *1* ? C'est toujours hors programme ! En fait, tout caractère manipulable par l'ordinateur (lettres, chiffres, ponctuations…) est associé à un nombre entier compris entre 0 et 255 appelé code ASCII. Pour convertir un caractère en une suite de *0* et de *1*, on prend le code ASCII du caractère et c'est ce code, exprimé en base 10, qui est converti comme précédemment en base 2.
Les codes ASCII des lettres *T, e, c, k* et *l* sont respectivement 84, 101, 99, 107 et 108. 84 en base 10, c'est 01010100, 101 en base 10 c'est 01100101 en base 2… d'où les *0* et *1* donnés ci-dessus.
Pour retrouver le texte initial, on prend les bits par paquets de 8, on converti chaque paquet en base 10 et on cherche la lettre ayant ce nombre comme code ASCII.

[2] Là, je ne vous expliquerai pas comment j'ai fait. Je peux juste vous avouer que j'ai eu du mal et que j'ai employé le mode de représentation du type *real* dans le langage Turbo pascal version 7 datant de 1992. Il y a peu de chances que ce format qui est employé dans VB 6 !

Cette notion d'instruction en contenant d'autres n'est pas neuve : on la retrouve dans le test (*si*) et les boucles (*pour*, *répéter* et *tant que*).

Une instruction complexe ne peut pas être comprise directement par l'ordinateur. Impossible donc de la traduire directement en *0* et *1* sans l'étape préalable de compilation.

1 C. Différence de fonctionnement entre l'ordinateur et nous

Il ne s'agit pas de faire un cours de matériel : ce qui m'intéresse ici, c'est uniquement le processeur, et encore : pas question d'étudier finement son fonctionnement réel, mais juste de comprendre *grosso modo* son principe.

Le processeur exécute réellement des instructions, mais ces dernières ont peu à voir avec celles dont nous parlons. Lorsque vous parlez à un très jeune enfant, vous employez un vocabulaire très simplifié et des tournures de phrase élémentaires : faire à un bambin de deux ans un discours fleuve, plein d'ironie, d'implicite, de sous-entendus et de phrases compliquées aura deux conséquences possibles : soit vous le ferez pleurer, soit vous le ferez rire, mais en tout cas, il est certain qu'il ne comprendra rien.

Le processeur obéit à la même logique : il ne *comprend* pas ce qu'il fait, car il n'a, au sens propre, aucune intelligence. On lui donne un ordre, il l'exécute, point, sans se poser la question de sa pertinence ou de sa validité. Les instructions seront donc élémentaires et très simples. Lorsque je dis à mes étudiants « vous pouvez faire une pause », il faudrait que je traduise cela au processeur en détaillant énormément : « posez vos crayons, levez-vous, prenez votre manteau, enfilez-le, marchez vers la porte, ouvrez-là… ».

Voyez bien le dilemme :

- la phrase « vous pouvez faire une pause » est très claire pour nous, mais n'est pas compréhensible pour quelqu'un ne comprenant pas la notion de pause ;
- l'autre version avec plein de détails peut être mise en œuvre par tous puisqu'il n'y a que des instructions de base[1]. Le problème, c'est que nous obtenons quelque chose de beaucoup plus verbeux, et, paradoxalement, moins intelligible pour nous puisque cette formulation nous fait perdre de la sémantique : l'idée que l'on fait un pause disparaît au profit d'une suite d'opérations élémentaires dont on ne voit pas la finalité.

Je résume :

- l'ordinateur (le processeur) ne comprend que des instructions élémentaires. Pour faire une instruction au sens où nous l'entendons, il faudra plusieurs, voire des dizaines d'instructions élémentaires qui ne sont pas très lisibles par le développeur ;
- le développeur, au contraire du processeur, est intelligent et est doué d'abstraction. Les instructions qu'il manipule sont donc de plus haut niveau et sont incompréhensibles par le processeur.

Dit autrement, nous avons deux points de vue :

- le processeur ne comprendra que des instructions élémentaires qui pourront, elles, être codées sous la forme de *0* et de *1* ;
- le développeur souhaite disposer d'un langage de programmation qui soit proche de lui et de sa façon de raisonner. Que cela ne corresponde pas à la logique de l'ordinateur n'est pas son problème.

[1] Je suppose évidemment que les mots *crayon*, *manteau*, *enfiler*… sont connus.

1 D. La phase de compilation

C'est là tout l'enjeu des langages de programmation et de leur évolution : les premiers langages (assembleur par exemple) étaient très proches de l'ordinateur, mais pas du développeur. Ce langage était simpliste mais pas agréable à utiliser.

Ensuite, les langages plus évolués, dits procéduraux (le C, le Pascal…) ont vu le jour. Ils s'adressent exclusivement aux développeurs et apportent de l'abstraction qui permet d'écrire des programmes proches du raisonnement intellectuel.

C'est pour cette raison que le monde des langages de programmation est en perpétuelle évolution : les chercheurs tentent perpétuellement d'améliorer la qualité des langages pour qu'ils soient les plus proches possible de notre perception de la réalité. C'est dans cet ordre d'esprit que les langages modernes (Delphi, C++, Java, un petit peu VB…) sont orientés objet[1].

Le problème, c'est que plus le langage de programmation est proche de nous, plus il est aisé à utiliser pour nous, mais plus il s'éloigne du langage assembleur compris par l'ordinateur.

Pour passer de l'un à l'autre, à savoir du langage de développement au langage compris par le processeur, on utilise un compilateur. Vous aurez compris que plus le langage de développement s'améliore, plus le compilateur doit être sophistiqué pour combler le fossé qui s'élargit entre le langage du développeur et le langage assembleur.

Voici un petit schéma récapitulant les choses :

Le problème de notre vocabulaire actuel, c'est que l'on ne cesse de parler de programmes. S'agit-il du résultat de la compilation ou du programme écrit par le développeur ?

Pour préciser les choses, nous allons introduire un peu de vocabulaire :

Le programme *source* (ou source[2]) est le programme écrit dans le langage de programmation choisi par le développeur.

Lorsque le source est compilé, on obtient le programme *exécutable* (ou exécutable) qui est directement compréhensible par le processeur.

Notre schéma précédent peut donc se réécrire ainsi :

[1] Cette notion d'objet sous VB est hors sujet pour nous.

[2] Ce sera *le* source (pour programme source) et non ~~*la*~~ source.

2. Situons VB dans ce cycle de développement

Vous vous sentez peut-être un peu frustré, car vous venez pour écrire du code et, depuis quatre pages, je vous sers un discours plus ou moins clair.

Où veux-je en venir ? Vous serez d'accord avec moi pour dire que le rôle du compilateur tel que je vous l'ai présenté est crucial. Nous allons donc voir où cet outil se situe dans VB.

VB, comme tous les langages modernes, dispose d'un EDI (Environnement de Développement Intégré). Cela signifie que tous les outils dont vous avez besoin pour travailler avec le langage Basic sont intégrés dans VB. Ainsi, au sein même du programme VB, vous pouvez :

- écrire votre programme source (VB dispose donc de la fonction d'éditeur de texte) ;
- compiler votre source (VB intègre donc un compilateur) ;
- corriger vos erreurs de syntaxe grâce à l'intégration de l'éditeur et du compilateur ;
- faire tourner l'exécutable issu de la compilation du source ;
- exécuter le programme instruction par instruction, évaluer le contenu des variables… (VB dispose donc également d'un débuggeur, programme dont l'objet est d'aider à localiser les bugs[1]).

Vous voyez que VB est bien plus qu'un langage. C'est pourquoi j'ai parlé du *programme* VB et non du *langage* VB. Savoir cela est important pour bien distinguer les étapes lorsque vous travaillez. Nous étudierons ces différentes fonctions dans le cours.

[1] Si nous nous plaçons dans le cadre automobile, un bug sera par exemple une voiture qui accélère quand vous débrayez. Un bug, c'est donc plus précisément une erreur de conception. La technicité des algorithmes fait que ce qui est impensable dans la majorité des domaines reste toléré, voire accepté en informatique. Cela dit, moins il y a de bugs, mieux c'est !

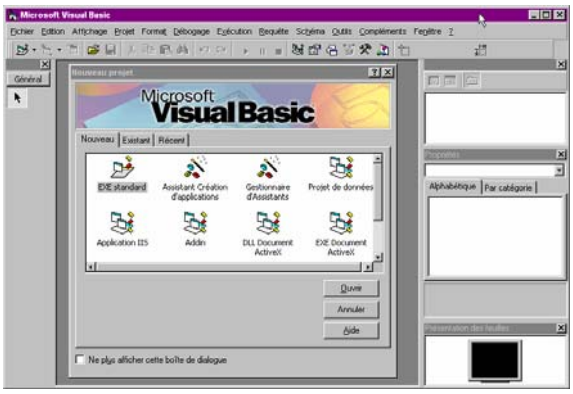
3. Interface VB

3 A. Vue d'ensemble

Tout au long de ce cours, nous allons étudier les différents éléments constituant VB, à savoir :

- les instructions VB, soit l'écriture du source ;
- comment exécuter le source tapé (et donc comment le compiler) ;
- comment débugguer le source s'il contient des erreurs.

Avant tout, lançons VB et découvrons l'affichage :



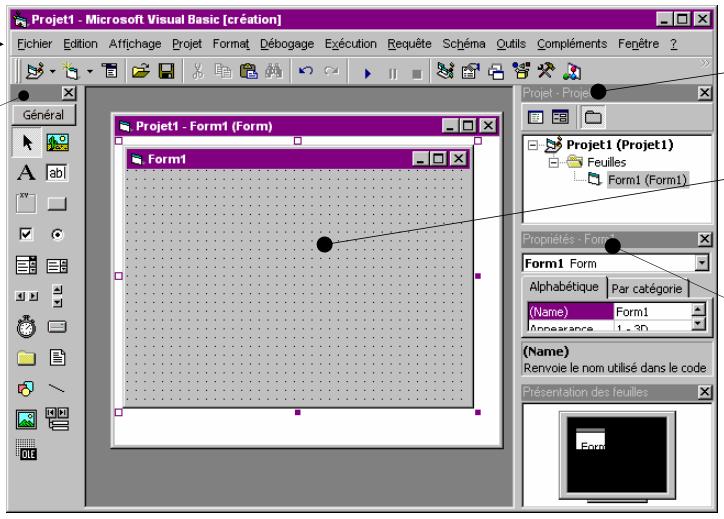
VB est lancé, mais vous avez en plein milieu de l'écran une fenêtre appelée *Nouveau projet* qui vous propose des options pas forcément très claires.

Vous devrez demander à votre meilleur ami (le bouton *Aide* présent sur la fenêtre) pour avoir des explications détaillées. Je vais juste vous dire l'essentiel, à savoir que cette fenêtre permet de choisir le type d'application que vous voulez réaliser : une *DLL*, un programme exécutable…

Vous pouvez (et je vous le conseille) cocher la case *Ne plus afficher cette boîte de dialogue* pour que VB choisisse automatiquement *EXE standard* qui est le seul choix à faire.

Le jour où vous voudrez écrire une DLL, vous pourrez toujours le faire en passant par la boîte de dialogue de la commande *Fichier/Nouveau projet*[1].

Barre de menus, commandes permettant d'agir sur le source (le menu Projet *permet d'ajouter des éléments au source,* Exécuter *permet de l'exécuter…).*



La boîte à outils *contient tous les contrôles regroupés par thèmes. Par défaut, seuls les contrôles généraux (les plus fréquents) sont affichés.*

Navigation dans le projet (feuilles et modules).

Fenêtre contenant la feuille (vierge pour le moment).

Propriétés du contrôle sélectionné (ici, la feuille).

[1] Sous-entendu menu *Fichier*, commande *Nouveau projet*.

Notez deux choses très importantes :

- vous pouvez avoir une interface légèrement différente de la mienne, soit parce que vous utilisez VB Net (qui vient de sortir alors que j'écris ces lignes), soit parce que vous avez une version de VB 6 différente de la mienne. Dans ce cas, il vous manquera peut-être certaines commandes et contrôles ou, au contraire, vous en aurez plus. Cela n'a aucune importance, car nous n'aborderons pas ces outils très pointus et totalement inutiles à notre niveau ;
- ne comptez pas sur moi pour vous présenter en détail l'interface graphique et toutes ses possibilités. Utilisez l'aide de VB[1] (touche *F1* ou menu *?*) pour découvrir sa richesse. Je ne parlerai que des menus et commandes essentielles.

3 B. Les éléments clés

3 B 1. Le projet

Un programme source VB est constitué de plusieurs fichiers. En effet, chaque feuille est stockée dans deux fichiers :

- la description de la feuille et de ses contrôles est dans un fichier d'extension *.frm* (pour *FoRMulaire*) ;
- le code source associé à cette feuille est dans un fichier d'extension *.bas* (pour *BASic*, le langage utilisé par VB). Notez dès à présent que les fichiers contenant le code s'appellent des modules et sont organisés de façon précise. Nous reviendrons bien entendu plus tard sur les modules.

Comme un programme peut très vite regrouper des dizaines de feuilles, vous imaginez le nombre de fichiers associés au même programme ! Pour les chapeauter tous, VB ajoute la notion de projet. Un projet est un contenant de tous les fichiers associés à un programme, un peu comme un dossier contenant divers fichiers. Concrètement, un projet sera un fichier d'extension *.vbp* (*VB Projet*) contenant la liste de tous les fichiers (feuilles et modules) à utiliser.

Dans la barre de titre de la fenêtre précédente (tout en haut de la fenêtre), vous voyez *Project1 – Microsoft Visual Basic [création]*. En lançant VB, vous êtes directement dans un projet vierge appelé *Project1*, contenant une seule feuille (*Form1*) sans code associé ni module. Si vous ajoutez des feuilles, elles seront automatiquement intégrées au projet.

Inversement, lorsque vous voulez ouvrir un programme déjà existant (pour modifier par exemple une feuille ou un morceau de code), ce n'est pas la feuille seule que vous allez ouvrir puisqu'elle ne sert à rien hors projet, mais le projet la contenant. Ouvrir le projet revient à accéder aux différents constituants, vous pouvez dès lors modifier la feuille que vous souhaitez, puis recompiler le programme.

Sans être trop technique, la notion de projet aide à mutualiser le code. Par exemple, vous pouvez définir une feuille et son code pour demander à l'utilisateur de rentrer un mot de passe. Si vous avez besoin de cette fonctionnalité d'authentification par mot de passe dans plusieurs applications, il vous suffira d'ajouter cette feuille aux différents projets concernés. Notez bien que cet ajout revient à insérer une ligne dans le fichier projet. Vous n'avez donc pas à dupliquer votre feuille de mot de passe. Ainsi, si vous devez la modifier, vous ne le modifiez qu'une fois et tous les projets l'utilisant seront automatiquement mis à jour. On retrouve le principe du raccourci sous Windows.

[1] Certaines versions de VB ne disposent pas de l'aide en ligne. Dans le cadre d'un développement personnel, il est impensable de s'en passer.

3 B 2. La barre de menus

Je vais vous présenter rapidement les différents menus.

Barre de menus

Fichier

Classique sous Windows, ce menu permet d'ouvrir, enregistrer, fermer ou imprimer des projets mais aussi d'enregistrer ou d'imprimer des modules ou des feuilles.

Édition

Toujours aussi classique, permet d'annuler ou refaire une action, de faire du couper, copier et coller, de chercher du texte… Les commandes de fin de ce menu permettent également d'accéder à des outils d'aide à la frappe du source.

Affichage

Classique… vous permet de définir les fenêtres à afficher. Dans d'autres applications, ce menu est souvent un sous-menu de *Édition*. Ce n'est pas le cas ici du fait de sa longueur. Il mérite bien un menu à lui tout seul.

Projet

Tout ce qui concerne la gestion du projet est ici :

- ajout ou suppression de feuilles ou de module au projet ;
- ajout de composants à la boîtes à outils pour bénéficier d'autre chose que les composants standards ;
- options liées au projet.

Il est possible d'ouvrir plusieurs projets à la fois, de même que sous Word, vous pouvez travailler sur plusieurs documents simultanément. Les ressources nécessaires pour la gestion d'un projet font qu'il n'est guère réaliste d'en ouvrir plusieurs à la fois.

Format

Ce menu vous aidera à aligner les contrôles sur vos feuilles.

Débogage

Lorsqu'un programme ne fonctionne pas comme il le devrait, vous devez vous creuser la tête… en utilisant un outil qui vous aidera à traquer les bugs, le débogueur. (Voir la séquence 6.)

Exécution

Pour exécuter votre application, ce qui est tout de même sa finalité.

Requête et Schéma

Pour l'accès aux bases de données, hors sujet pour nous.

Outils

Pour ajouter automatiquement l'en-tête d'un sous-programme, ce qui est d'un intérêt limité, mais aussi pour paramétrer le comportement du programme VB (éditeur, compilateur…).

Compléments

Sans objet pour nous.

Fenêtre

Menu classique sous Windows… permet de naviguer dans les différentes fenêtres ouvertes.

?

C'est l'aide… sans conteste votre meilleur ami !

3 B 3. Les barres d'outils

L'ergonomie de Windows (et des applications fonctionnant sous Windows) impose que :

- toutes les commandes pouvant être exécutées par une application soient présentes dans les différentes barres de menu ;
- parmi ces commandes, les plus utilisées soient présentes dans les barres d'outils et sont représentées par un icone[1] ;
- parmi les commandes des barres d'outils, les plus utilisées possèdent des raccourcis clavier.

Les nouvelles normes d'ergonomie Windows (qui sont en perpétuelle évolution) imposent que, dans la barre de menus, toute commande :

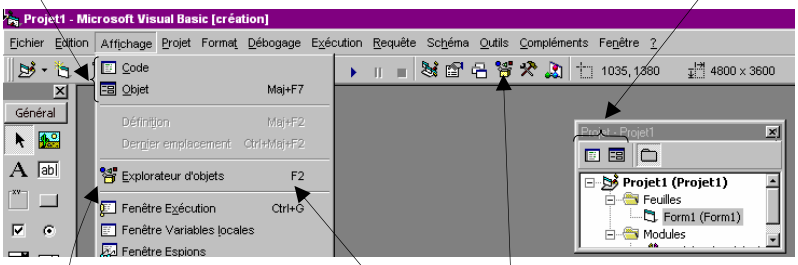
- disposant d'un icone dans une barre d'outils soit précédée du dessin de l'icone pour que l'utilisateur associe facilement les deux ;
- disposant d'un raccourci clavier soit suivie de ce raccourci pour que l'utilisateur le découvre.

Avec cette norme, évidemment suivie par VB, impossible de justifier l'usage régulier de la barre de menus en disant que vous ne saviez pas qu'il y avait un icone !

Prenons l'exemple du menu *Affichage* :

À quoi servent ces deux icones ?

Pour le savoir, cherchons-les dans les menus…
Ah, les voilà ! C'est donc afficher le code ou l'objet… c'est plus clair.



La commande est suivie d'un raccourci clavier indiquant qu'appuyer sur F2 *affiche l'explorateur.*

La commande est précédée d'un icone, ce qui signifie qu'une barre d'outils permet d'exécuter le programme. Où est cet icone ? Là !

Vous noterez que les commandes suivantes du menu *Affichage* (*Fenêtre exécution*, *Fenêtre espions*…) disposent d'un icone qui n'apparaît pas dans les barres d'outils. Est-ce une erreur de ma part ? Non, cela signifie juste que l'icone existe mais n'est pas dans les barres d'outils présentes à l'écran. En effet, il existe de nombreuses barres d'outils, qui ne sont pas toutes affichées et sont de surcroît personnalisables (voir la commande *Affichage/Barres d'outils…*).

La façon la plus facile d'utiliser une commande, c'est, dans l'ordre :

- d'aller dans la barre de menus car la commande est écrite en toutes lettres : *Affichage*, c'est explicite !
- d'aller dans la barre d'outils (si la commande y est présente) car un petit dessin nous rappelle sa fonction. Parfois, il est très intuitif (l'imprimante pour imprimer), parfois,

[1] Attention à l'erreur classique qui ne fait pas très sérieux : il faut bien distinguer *une* icône (peinture religieuse sur un panneau de bois) de *un* icone, dessin représentant ce qu'il désigne. En informatique, nous manipulons bien des icones : le bouton pour imprimer symbolise une imprimante…

il est moins limpide : pour exécuter le programme, la convention est d'utiliser le symbole *play* (marche) des magnétoscopes et lecteurs hi-fi, à savoir le petit triangle. C'est évident… quand on le sait (allez voir dans le menu *Exécution* pour identifier l'icone). Quant à l'icone de l'explorateur d'objets…

– d'utiliser le raccourci clavier s'il existe. C'est le moins simple car le raccourci est souvent arbitraire. Pourquoi *F2* pour afficher l'explorateur d'objets ? Eh bien ma foi, pourquoi pas ?

La façon la plus rapide d'utiliser une commande, c'est, dans l'ordre d'utiliser :

– le raccourci clavier s'il existe. Une ou deux touches à appuyer, et hop, la commande est appliquée ;
– l'icone de la barre d'outils. Il faut lâcher le clavier, prendre la souris et faire un clic ;
– la commande de la barre de menu, car il faut aller dans le bon menu (un clic) puis lancer la commande (un autre clic).

Résumons : icones et raccourcis clavier sont là pour vous simplifier la tâche ; servez-vous en !

3 C. Le travail sous VB

Une façon de voir les choses, c'est de vous dire que programmer sous VB revient à :

– pêcher dans la palette des composants les éléments (contrôles ou autre) dont on a besoin ;
– écrire du code ;
– exploiter quelques icones pour tester son programme puis l'enregistrer.

Évidemment, ces différentes étapes ne sont parfois pas si simples.

3 D. Ce n'est pas fini

J'insiste, à ce stade, vous ne savez rien faire et l'interface VB doit demeurer assez nébuleuse. Pour vous l'approprier, testez les différentes commandes pour voir ce qu'elles font et utilisez l'aide.

Une façon simple de recourir à l'aide pour un élément précis :

– dans un menu, mettez votre souris sur la commande qui vous intéresse (sans cliquer) et faites *F1*. Vous aurez directement l'aide de cette commande ;
– de même, mettez votre souris sur un contrôle de la boîte à outils et faites *F1*… vous obtiendrez une explication détaillée.

Séquence 3

Un 1er programme sans prétention

Nous allons découvrir l'interface du programme VB et tout ce que ce programme permet de faire. Pour cela, nous allons écrire un programme.

Contenu

Les différentes phases dans l'écriture et l'exécution d'un programme
L'interface utilisateur de VB
Les commandes principales

Capacités attendues

Maîtriser les notions clés de source, exécutable et compilation
Savoir manipuler les commandes essentielles de VB
Utiliser l'aide

1. Présentation

1 A. Objet de la séquence

Vous avez compris que nous allons enfin écrire un programme. Stop au baratin intéressant mais improductif, place à l'écriture de code !

L'objet de ce programme est triple :

- apprendre concrètement comment utiliser VB pour développer une application ;
- découvrir les contrôles Windows et leur mode de fonctionnement (modèle objet) ;
- écrire du code pour découvrir le fonctionnement événementiel de VB et les instructions VB correspondant à l'algorithmique que vous connaissez.

Cette séquence très courte est juste une sensibilisation à ces trois concepts clés (le modèle objet, la programmation événementielle et le langage de VB). Les séquences qui suivent rentreront dans le détail.

1 B. Le programme que l'on va réaliser

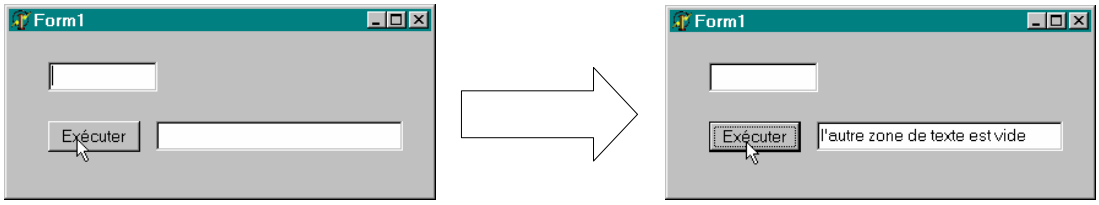
1 B 1. Description

Nous allons créer une application possédant deux zones de texte et un bouton. Lorsque l'on clique sur le bouton, deux situations se présentent :

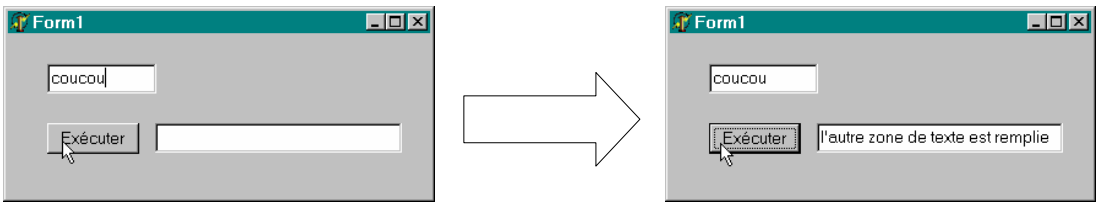
- si la première zone de texte est vide, la seconde contiendra le texte *L'autre zone de texte est vide* ;
- si la première zone de texte contient quelque chose, la seconde contiendra *L'autre zone de texte est remplie*.

1 B 2. Illustration

Première situation :



Seconde situation :



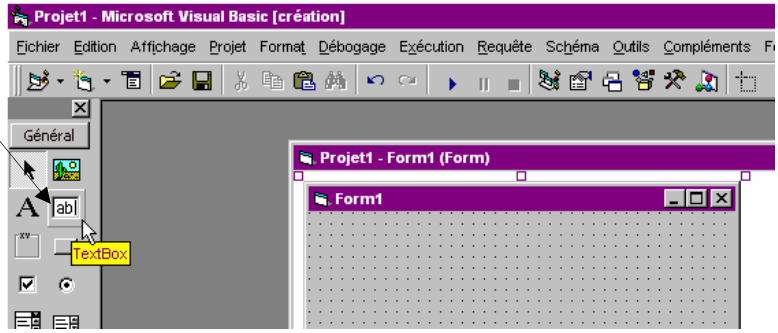
2. Réalisons l'application

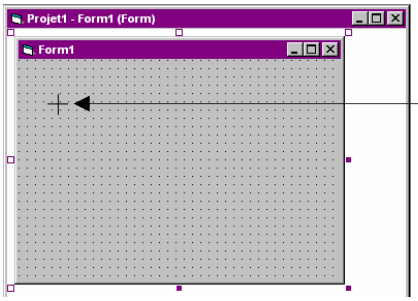
2 A. Création de la feuille (réalisation de l'interface)

Une fois VB lancé, accédez à la feuille *Form1* (ouverte automatiquement), diminuez sa longueur et sa largeur avec la souris (la feuille est une fenêtre Windows comme une autre) puis placez deux zones de texte et un bouton dessus.

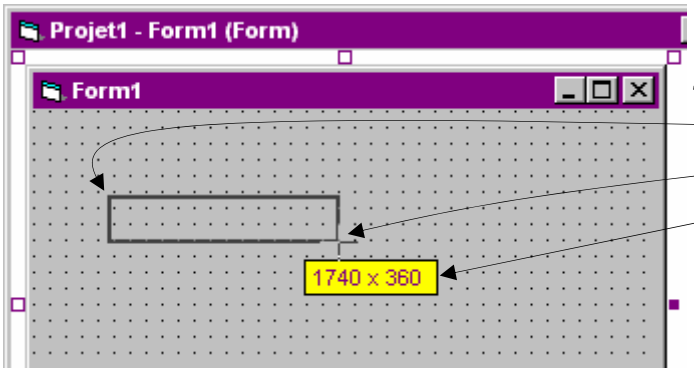
Comment ? C'est tout simple : vous cliquez dans la boîte à outils pour sélectionner le contrôle que vous voulez, puis vous dessinez le contrôle dans la feuille là où vous souhaitez qu'il soit placé. Euh, dessiner ? Oui : vous placez votre curseur souris devenu une croix sur la feuille, vous cliquez et laissez appuyé et vous déplacez le curseur pour définir la taille du contrôle.

1. Je clique sur le contrôle (c'est ce contrôle qui est la zone de texte).



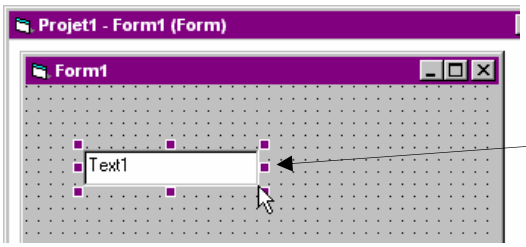


2. Je me place sur la feuille, là où la zone de texte doit apparaître. Notez que mon curseur est une croix.



3. Je clique, je déplace la souris de ici vers là…

J'obtiens la taille en Twips *de mon contrôle.*



4. … et je lâche. Ma zone de texte est créée.

Qu'y a-t-il dans ma zone de texte ? Le mot *Text1*. À quoi sert-il ? C'est le nom de ma zone de texte, composé de deux parties :

- *Text* (pour *TextBox*), c'est le nom VB de ce que les règles d'ergonomie Windows appellent zone de texte. L'aide de VB sur *TextBox* indique d'ailleurs que c'est un contrôle de saisie monoligne par défaut[1], définition savante de la zone de texte ;
- *1* (pour… un) indique que c'est la première zone de texte de la feuille.

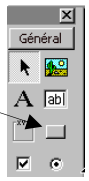
Vous savez qu'en programmation tout élément (variable, type, procédure ou fonction) possède un nom. L'intérêt, c'est qu'il suffit d'utiliser son nom pour accéder à un élément[2].

Les contrôles n'échappent pas à la règle : pour les manipuler, il faut leur donner un nom. D'ailleurs, notre application contiendra deux zones de texte. Nous sommes donc obligés d'avoir deux noms distincts pour pouvoir accéder à l'une ou à l'autre.

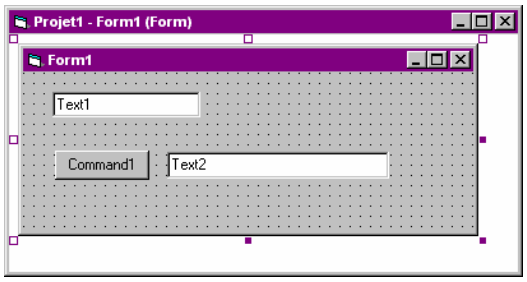
Lorsque vous déclarez des variables, la syntaxe vous oblige à mentionner son nom et son type. Ici, lorsque l'on ajoute un contrôle, toute la manipulation se fait à la souris, aucun nom ne vous est demandé. C'est pourquoi VB utilise un nom par défaut (*Edit1*) histoire que tout composant en possède un. Certes, *Edit1* est plus parlant que *Teckel* ou *ARR778M_U*. Au moins, on sait que c'est une zone de texte. Mais c'est comme si la variable entière contenant votre âge était nommée *Entier1*…

Bref, le nom du contrôle proposé par défaut n'est pas acceptable. Lorsque l'on a posé les différents contrôles sur la feuille, il faut se dépêcher de les renommer. Nous ferons cela tout à l'heure.

Pour le moment, ajoutez une seconde zone de liste et un bouton (il est ici, juste en-dessous de la zone de texte).



Déplacez ensuite les contrôles[3] de façon à obtenir ceci :



Vous aurez noté les noms des contrôles : *Text1*, *Text2* et *Command1*[4]. Nous allons comme promis les renommer.

Ici, il ne sera pas évident de trouver des noms beaucoup plus parlants puisque l'application est sommaire. Si nous avions un bouton permettant de quitter l'application, on l'appellerait *Quitter*. De même, si une zone de texte servait à saisir une adresse, nous l'appellerions

[1] Ceci signifie que si vous placez une zone de texte sur une feuille, elle sera monoligne donc ne permettra de saisir qu'une (éventuellement très longue) ligne de texte. Si vous voulez modifier ce comportement et pouvoir saisir plusieurs lignes, vous devrez paramétrer le contrôle (nous verrons cela).

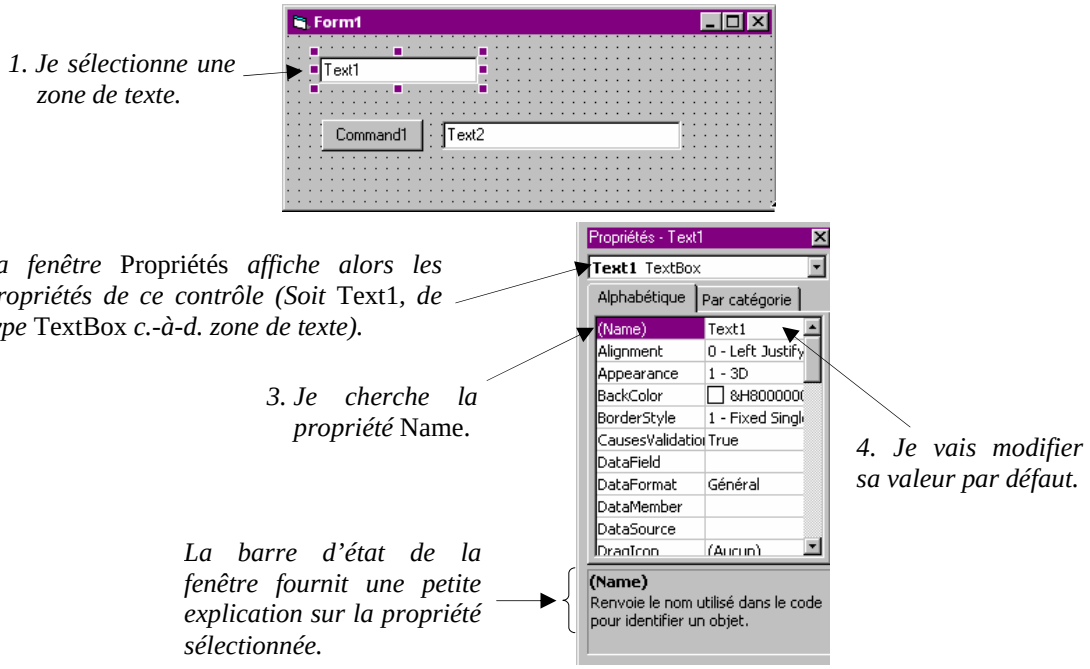
[2] C'est pourquoi nous avons tous un nom et un prénom qui nous identifient. Comme cette identification n'est pas parfaite (homonymes), il en existe d'autres : l'identifiant fiscal unique, le numéro de Sécurité sociale, votre numéro de client chez EDF…

[3] Classique sous Windows : vous cliquez sur un contrôle de la feuille, vous maintenez appuyé et vous pouvez déplacer le contrôle. Quand vous lâchez le bouton de la souris, le contrôle est positionné où vous êtes.

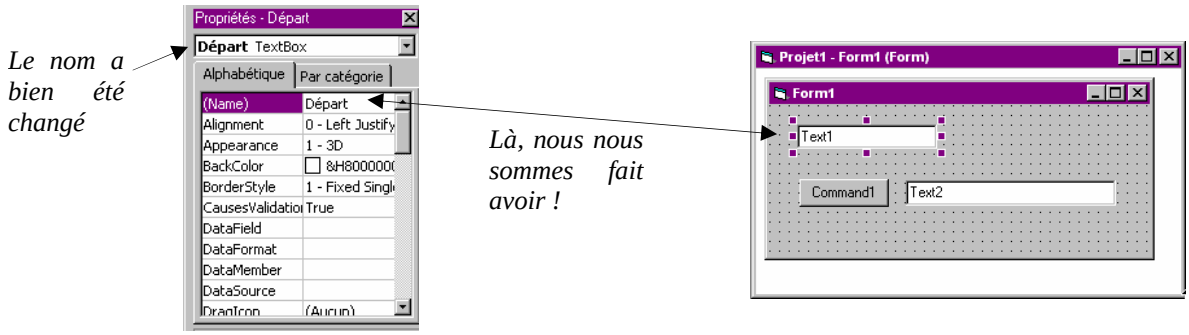
[4] *Command1*, c'est pour *CommandButton* numéro 1.

Adresse[1]. Ici, j'appellerai le bouton *Exécution*[2]. Quant aux zones de texte, je vous propose *Départ* et *Arrivée*.

Comment les renommer ? C'est très simple : vous les sélectionnez un par un dans la feuille en cliquant dessus, puis vous allez dans l'inspecteur d'objets. Dans l'onglet *Propriétés*, vous cherchez la propriété[3] *Name* (elles sont en anglais et dans l'ordre alphabétique) et vous remplacez la valeur par défaut (en bleu) par le nom choisi :



Une fois la valeur modifiée, on obtient :



Il semblerait que ce qui est affiché dans la zone de texte ne soit pas son nom puisque je l'ai changé et *Départ* (nouveau nom de ma zone de texte) contient toujours fièrement *Text1*.

✎Exercice 1✎

Nous allons régler le problème… dès que possible. En attendant, changez le nom du bouton en *Exécuter* et celui de la seconde zone de texte en *Arrivée*.

[1] Il existe une pratique consistant à inclure le type du contrôle dans son nom. Par exemple, tous les boutons commenceraient par *B* et les zones de texte par *ZDT*. Ainsi, on n'utiliserait pas *Quitter* et *Adresse* mais *BQuitter* et *ZDTAdresse*. Je n'y suis pas favorable car l'écriture est alourdie et, si votre interface est correcte, il ne doit pas y avoir d'ambiguïté : *Quitter*, qui est un verbe, ne doit faire référence qu'à un bouton. *Adresse* doit logiquement correspondre à un contrôle permettant de saisir une chaîne de caractères, à savoir une zone de texte.

[2] VB accepte l'utilisation des caractères accentués dans les identificateurs (nom de variable, de sous-programme…). Nous serions bien bêtes de ne pas en profiter !

[3] Nous découvrirons cette notion de propriété dans la séquence suivante.

✎Exercice 2✎

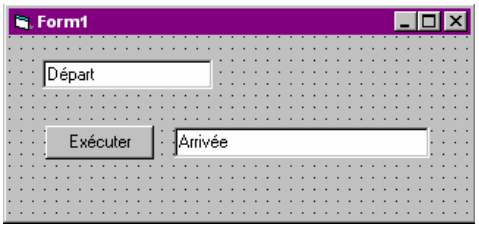
En sélectionnant le bouton, cherchez sa propriété *Caption*. Observez sa valeur. Essayez de mettre *Teckel* dans *Caption* pour voir ce qui se passe dans la feuille.

✎Exercice 3✎

Le problème a été réglé pour le bouton. Cherchez une propriété *Caption* associée aux zones de texte. Si vous ne la trouvez pas, lisez l'aide sur *Caption* puis tentez de chercher une propriété de la zone de texte *Départ* contenant *Text1* et changez-la en *Départ* (idem pour *Arrivée*).

Bien. Continuons notre programme (remettez *Caption* à *Exécuter*.

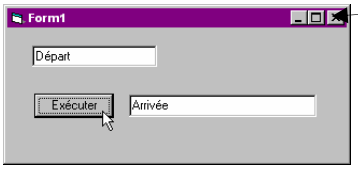
Nous avons donc la feuille suivante :



2 B. Test de l'application (1)

Exécutez votre application. Comment ? Eh bien, menu *Exécution* (c'est le moment d'assimiler l'icone voire le raccourci clavier de la commande *Exécuter*).

Voici ce que l'on obtient :



Pour terminer le programme, il faut fermer sa fenêtre donc cliquer ici

Cette application a deux problèmes majeurs (à ce stade, ce ne sont plus des bugs) :

- il y a *Départ* et *Arrivée* d'écrit dans les zones de texte. Or, je voudrais que les zones soient initialement vides et que l'utilisateur puisse taper ce qu'il veut dedans ;
- dans ma copie d'écran, le curseur est sur le bouton. Ce n'est pas par accident : j'ai frénétiquement essayé de cliquer dessus, mais il ne se passe rien.

✎Exercice 4✎

Pouvez-vous me dire l'origine de chacun de ces problèmes ? Essayez de résoudre le premier.

2 C. Corrigeons l'interface

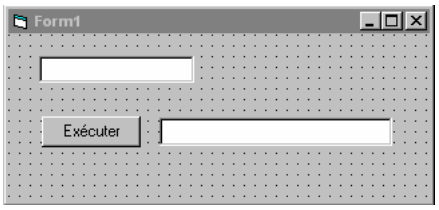
2 C 1. La manipulation

✎Exercice 5✎

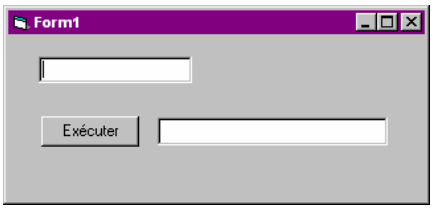
Réglez le problème du contenu des zones de texte (la solution est dans la correction de l'exercice précédent).

Voici le nouvel aspect de la feuille, en mode conception puis exécution.

Mode conception



Mode exécution



Les zones de texte sont vierges. Ainsi, ce n'était pas leur nom qu'elles contenaient, mais bien leur valeur (leur contenu en somme), sachant que, par défaut, leur contenu était égal à leur nom.

Si vous n'avez pas compris cette phrase, relisez-là.

2 C 2. Point théorique

Il est sans doute bon de faire un point concernant les propriétés *Name*, *Caption* et *Text*. Vous avez peut-être l'impression que *Caption* et *Text* sont équivalentes. Ce n'est pas le cas :

- *Name* est une variable[1] contenant le nom du contrôle ;
- *Caption* est une variable contenant ce qui est écrit dans le bouton. L'utilisateur de votre programme ne peut pas modifier ce texte (le bouton ne s'édite pas) ;
- *Text* est une variable contenant le… contenu d'une zone de texte. Lorsque l'utilisateur modifie le contenu d'une zone de texte (c'est la finalité de ce contrôle), le programmeur récupère la valeur saisie en lisant la variable *Text*.

Dit autrement, *Caption* définit une valeur fixe, comme si vous graviez votre nom sur votre porte d'entrée. Au contraire, *Text* reflète une valeur changeante (par exemple le numéro de la page que vous lisez).

Vous me trouvez sans doute pointilleux mais cela est très important à assimiler puisque ce sont ces détails, et bien d'autres, qui font la qualité de l'interface graphique.

2 D. Écrivons le code

2 D 1. L'événement

Bien. Arrivé là, votre interface est correcte (les zones de texte sont initialement vides), mais il ne se passe toujours rien lorsque l'on clique sur le bouton.

Nous allons donc devoir dire ce qui doit se passer lorsque l'on clique sur le bouton.

Ce qui doit se passer, c'est facile : nous utiliserons une alternative classique (le *si* algorithmique).

En revanche, comment dire que notre alternative doit être exécutée lorsque l'on clique sur le bouton ? Cela, c'est de la programmation événementielle. Nous verrons cela en détail dans la séquence suivante. Pour le moment, je vais juste vous dire que l'on peut écrire du code qui sera exécuté lorsque des événements précis arrivent.

Par exemple, le fait de cliquer sur un bouton est un événement (il se passe quelque chose dans le monde palpitant de l'interface graphique).

[1] Pour le moment, une propriété n'est rien de plus qu'une variable.

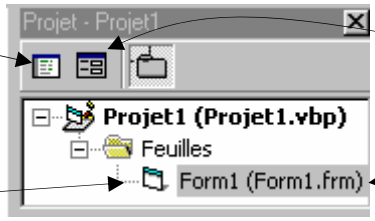
Pour écrire du code, nous devons ouvrir la fenêtre contenant du code. Oui, c'est un peu évident, mais bon… L'idée est qu'actuellement, vous avez sous les yeux uniquement la feuille. Or, la feuille, c'est du graphisme ; il n'y a pas de place pour le code.

Il y a trois façons pour afficher la fenêtre du code :

- passer par la commande (nous l'avons vue incidemment) ;
- utiliser l'icone ;
- faire un clic droit sur un des contrôles et choisir *code* (c'est la solution la moins rapide).

La meilleure solution, c'est l'icone. Je vous rappelle qu'elle se trouve dans la fenêtre *Projet* :

Cliquer ici affiche la fenêtre code de l'objet sélectionné là



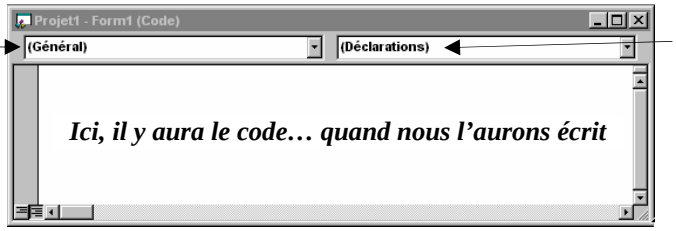
Cliquer ici affiche l'objet sélectionné là

Cette fenêtre *Projet* n'est pas grosse mais elle est cruciale. En effet, toute la navigation dans le projet se fait à partir d'elle :

- lorsque vous chargez un projet, vous n'avez rien à l'écran. Il faut passer par cette fenêtre pour sélectionner l'objet (feuille ou module[1]) que vous souhaitez visualiser ;
- en conception, vous passerez constamment d'une feuille à une autre, d'un bout de code à un autre grâce à elle ;
- vous aurez noté que les deux boutons présentés représentent une bascule puisque l'un permet d'afficher un objet et l'autre le code lié à cet objet.

La feuille étant sélectionnée, j'ai appuyé sur le bouton affichant le code. Voici ce qui est apparu :

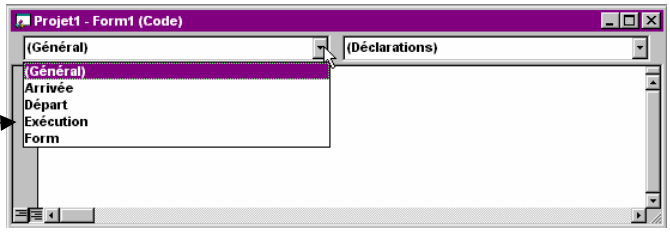
Cette zone de liste permet d'accéder à tous les contrôles de la feuille, elle comprise.



Cette zone de liste permet d'accéder aux différents événements du contrôle choisi dans la précédente zone.

Pour voir les différents événements qui peuvent arriver à un bouton[2], vous allez d'abord sélectionner le bouton *Exécuter* dans la liste de gauche :

Comme promis, voici les différents contrôles qui existent sur la feuille. Je vais choisir mon contrôle Exécution.

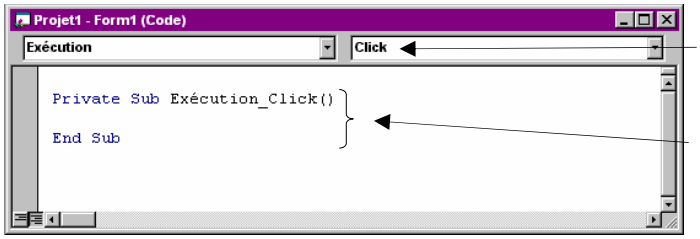


2 D 2. La procédure événementielle

En cliquant sur *Exécution*, il se passe quelque chose d'assez curieux : ma fenêtre se remplit (enfin, remplit est sans doute excessif) de code, comme l'atteste la copie d'écran qui suit.

[1] Module ? terme technique… nous le verrons plus tard.

[2] Nous verrons que ces événements sont définis par Windows. Impossible d'en ajouter. On peut juste espérer que tous ceux dont nous avons besoin sont bien présents.



Ce n'est pas moi qui au choisi Click, *il s'est mis tout seul.*

Je n'ai pas non plus tapé ce code.

Expliquons ce qui s'est passé.

Lorsque vous choisissez un contrôle, c'est forcément parce que vous comptez écrire du code événementiel lié à ce dernier. VB a une politique d'assistanat qui peut ne pas plaire, mais que l'on ne peut pas éviter. Le principe est simple : lorsque vous sélectionnez un contrôle, il se permet de préparer la procédure de l'événement principal de ce contrôle.

Je sens que vous vous interrogez sur cette procédure, allant même jusqu'à vous demander si vous n'avez pas loupé une page du cours. En fait, tout va bien : je n'ai pas encore parlé de cette procédure car c'est VB qui vient de la définir et c'est vous qui allez devoir écrire son contenu. On parle de procédure événementielle car c'est une procédure qui sera exécutée en réponse à un événement.

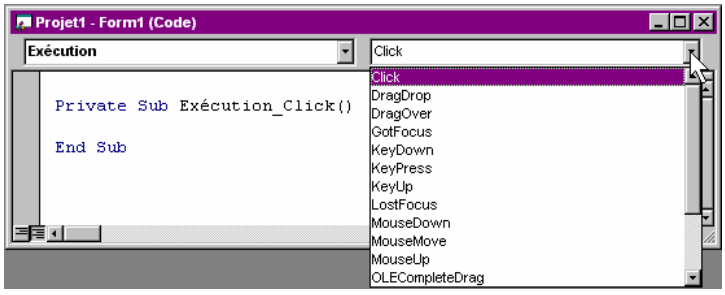
Vous comprendrez tout cela plus facilement dans la séquence suivante lorsque je vous aurai *briefé* sur les événements. Je veux juste dire que l'objet d'un bouton, c'est que l'on clique dessus. VB se dit avec raison qu'il y a de grandes chances que vous souhaitiez écrire le code lié au clic, et pas celui correspondant au survol souris.

C'est pourquoi il vous met l'en-tête de la procédure correspondante.

Cette copie d'écran nous indique que du code est associé à l'événement *click* du contrôle *Exécution*. En clair, lorsque vous cliquerez sur le bouton, le code sera exécuté.

Notez le nom de la procédure événementielle : *Exécution_Click*. Elle est nommée automatiquement par VB selon une convention tout à fait raisonnable : les noms du contrôle et de l'événement séparés par un tiret.

En déroulant la seconde zone de liste, on accède à tous les événements existant pour le contrôle choisi :



Visiblement, il n'y a pas que le click *! Si vous choisissez un événement, sa procédure événementielle sera définie.*

Attention, une procédure événementielle est avant tout une procédure. Si je la dis événementielle, c'est parce qu'elle est déclenchée par l'arrivée d'un événement, mais c'est avant tout une procédure comme une autre. Dit autrement, tout procédure devient événementielle si vous la liez à un événement.

Cela vous permet de voir par la bande que la déclaration d'une procédure ressemble à ce que l'on voit en algorithmique : au lieu du mot clé *procédure*, on met *sub* (on y gagne, c'est plus court). Vient ensuite son nom puis ses paramètres.

D'ailleurs, quels paramètres mettre ? Nous ne nous en préoccupons pas, c'est VB qui les définit. Ils varient en fonction de l'événement (le *click* ne possède aucun paramètre).

Ah, je n'ai pas parlé du mot *private*. Nous y reviendrons.

Votre curseur a été positionné entre les lignes *Private sub* et *End Sub*. Pourquoi ? Car c'est ici que nous allons écrire le code.

Il faut comprendre cela car d'une part c'est simple et d'autre part vous devez maîtriser un minimum votre code. Cela dit, tout est réalisé automatiquement par VB, donc pas de souci !

2 D 3. Testons l'application

✎Exercice 6✎

Comme la procédure événementielle est définie, vous pouvez tester votre programme en l'exécutant. Que se passe-t-il ?

2 D 4. Écrivons enfin le code

Pour cela, refaites la manipulation du paragraphe 2D2 afin de remettre *Exécution_Click*.

✎Exercice 7✎

Vous connaissez la propriété permettant d'accéder (en lecture ou en écriture) au contenu d'une zone de texte. Vous connaissez la façon d'accéder à un champ d'un enregistrement (opérateur « . »).

Sachant que vous pouvez considérer un contrôle comme un enregistrement et la propriété comme un champ, écrivez le code algorithmique (un *si*) faisant ce que je veux.

Il faut maintenant transcrire le code algorithmique en code VB. Là, c'est moi qui vais tout faire puisque vous ne connaissez pas encore le langage.

Version algorithmique :

```
si Depart.Text = ""
   alors
      Arrivee.Text := "l'autre zone de texte est vide"
   sinon
      Arrivee.Text := "l'autre zone de texte est remplie"
fin-si
```

Version VB :

```
01 if Départ.Text = "" then
02       Arrivée.Text = "l'autre zone de texte est vide"
03    else
04       Arrivée.Text = "l'autre zone de texte est remplie"
05 End If
```

Que faut-il retenir ? Nous verrons toutes les instructions VB dans la séquence 5. Notons juste ici que :

- ligne 1, le mot clé *then* doit être sur la même ligne que le *if*, sinon VB râle (la ligne devient rouge pour indiquer une erreur de syntaxe) ;
- lignes 2 et 4, l'affectation sous VB se note « = » ;
- VB possède des terminateurs d'instruction (*End If* ligne 5).

✎Exercice 8✎

Écrivez le code VB dans la procédure événementielle (entre le *Private sub* et le *End Sub*). Testez le programme. Cela doit marcher.

3. Remarque très importante

3 A. Posons le problème

✎Exercice 9✎

Modifiez le code de la procédure événementielle pour avoir :

```
Private Sub Exécution_Click()
   If Départ.Text = "" Then
         Arrivée.Text = Jean - Yves * Février
      Else
         Arrivée.Text = Aime * Les * Teckels
   End If
End Sub
```

Testez le programme. Curieux, non ? Une explication ?

3 B. Expliquons le problème

VB veut bien faire. C'est un maladroit qui veut bien faire. Son idée est de décharger au maximum le fardeau du pauvre développeur. Il ne demande donc pas à ce dernier de déclarer les variables utilisées, suivant en cela le fonctionnement des antédiluviens basics.

De prime abord, cela semble merveilleux : VB déclare pour vous toutes les variables, plus de soucis de type…

En fait, ce n'est pas merveilleux pour deux raisons :

- si vous ne déclarez pas une variable, VB va la déclarer pour vous lorsqu'il traitera une instruction la contenant. Il essayera de déterminer son type au mieux, mais cela ne sera pas parfait. Votre application ne sera donc pas pleinement efficace. Mais ce n'est pas encore le plus grave ;
- le plus grave, ce sont les fautes de frappes. Supposons que vous ayez une variable *TauxTVARéduit* dans votre programme. Supposons aussi que, par accident, vous l'écriviez une fois *TauxTVAReduit*[1].Lorsque VB arrivera sur l'instruction correspondante, il va se dire « tiens, une nouvelle variable. Bonjour, nouvelle variable. Je vais te déclarer pour le gentil programmeur. » Bref, là où vous croyiez faire un traitement sur votre variable *TauxTVARéduit*, VB le fera sur une autre variable qui n'a rien à voir. Le problème, c'est que cette erreur est très pernicieuse : vous verrez que votre programme bug, donc vous chercherez une erreur de logique (d'algorithme) mais pas une mauvaise écriture, à un moment donné, d'une variable.

Imaginez maintenant que vous deviez déclarer les variables. Arrivé à *TauxTVAReduit*, VB se dirait « tiens, une variable non déclarée ; j'avertis le gentil programmeur et je m'arrête. ». Vous voyez donc immédiatement qu'il s'agit d'une faute de frappe et vous la corrigez.

Dans le programme ci-dessus, *Jean*, *Aime*, *Février*, *Teckels*, *Yves* et *Les* sont des variables non déclarées donc automatiquement référencées par VB et initialisées à 0.

[1] Vous voyez la différence ? J'ai changé le « é » en « e ».

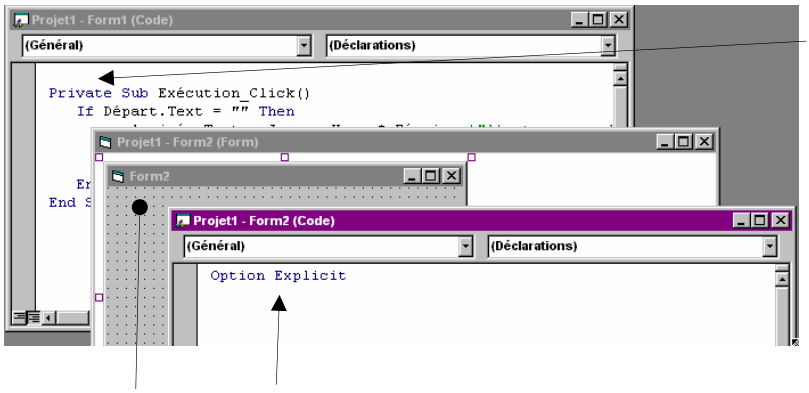
3 C. Résolvons le problème

Tout cela pour dire que je vous conseille très très fortement de vous obliger à déclarer les variables. Comment ? En allant dans *Outils/Options…*, onglet *Éditeur*, cochez la case *Déclaration des variables obligatoire*.

Tous les nouvelles fenêtres de code contiendront l'instruction magique *Option Explicit* obligeant le développeur à déclarer explicitement ses variables.

Le problème, c'est que cette modification ne s'applique pas au code déjà écrit.

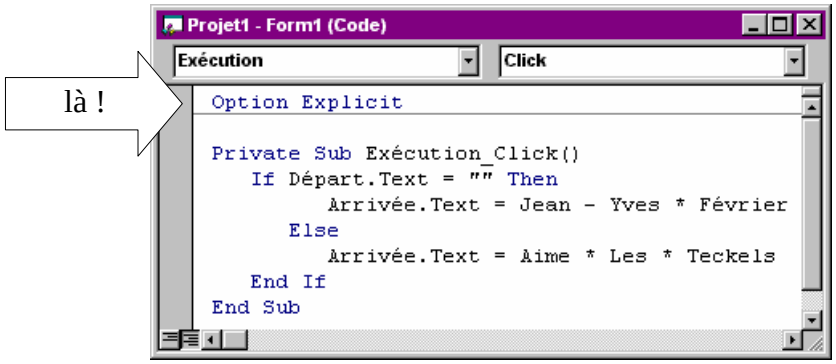
Démonstration :



Mon code actuel ne contient pas la ligne magique.

J'ai ajouté une nouvelle feuille à l'application (commande Projet/Ajouter une feuille*, j'ai choisi* Form*). Son code associé contient spontanément* Option Explicit*.*

Je dois donc l'ajouter dans le code :



Maintenant, si vous exécutez le code, VB vous indiquera un beau message d'erreur. En effet, il n'acceptera pas de traiter les variables *Jean*, *Aime*, *Février*, *Teckels*, *Yves* et *Les* puisqu'elles n'ont pas été déclarées.

Séquence 4

Objets et événements

Dans la séquence précédente, nous avons réalisé un petit programme sommaire mais exploitant l'ensemble des éléments de VB.

Nous allons maintenant approfondir ces concepts.

Contenu

Principe de base de la programmation objet.

Syntaxe.

Capacités attendues

Savoir utiliser des objets.

1. Le concept d'objet

Lisez très attentivement cette partie. Relisez-la. Imprégniez-vous-en. Pourquoi ? Parce qu'elle contient des réflexions assez générales mais qui sont très importantes pour la compréhension des objets.

1 A. Rôle général d'une application

Si l'on regarde les choses d'un peu haut, le rôle d'un programme informatique est toujours de manipuler de l'information.

Plus précisément, un programme exploite des informations pour en produire d'autres : il prend des données en entrée, réalise un traitement sur ces données et en fournit d'autres en sortie (c'est le résultat du programme).

Un exemple simple : le programme prend en entrée les chiffres d'affaires mensuels de l'entreprise et donne en sortie leur moyenne.

Un exemple un peu moins évident : un jeu, prenant en entrée les actions à la souris et au clavier du joueur et fournissant en sortie, suite à un traitement sur les entrées, le déplacement du joueur à l'écran et le son approprié.

Ainsi, que l'on soit face à un programme de gestion, à un jeu ou à un système d'exploitation, on retrouve constamment la notion d'information manipulée.

L'information étant cruciale, nous allons nous appesantir sur la façon dont un programme la représente.

1 B. Représentation de l'information

1 B 1. Variables d'un type de base

Nous venons de voir qu'un programme gérait de l'information. Nous allons maintenant étudier comment il la représente (stocke).

Les applications sont conçues pour l'homme. Je veux dire par là que les informations manipulées ont un sens pour les humains. Plus encore, ces informations sont des informations venant de notre monde : des âges, des noms, des prix, des clients, des factures…

La programmation structurée traditionnelle (langages Basic, C, Pascal…) manipule des variables typées (une variable est d'un type donné). Ces variables sont simples (elles n'ont qu'une valeur) et sont d'un des types de base suivants : entier, réel, chaîne de caractères, caractère ou booléen (les langages modernes rajoutent des types tels *date*…). Cela suffit pour manipuler les concepts simples du monde réel : les noms, les prix, les dates…

En revanche, lorsque l'on veut manipuler des informations plus complexes comme des clients, factures, commandes ou règlements, les types de base sont insuffisants : un client ne se limite pas à une chaîne de caractères ou à un entier !

Un autre exemple d'information complexe ? Eh bien, une zone de texte ! Vous avez vu dans la séquence précédente qu'une zone de texte possédait de nombreuses caractéristiques : son contenu, son nom, sa position…

Ces informations complexes, il faut pouvoir les représenter **correctement** et **efficacement**. La solution réside dans les types structurés.

1 B 2. Les types structurés

L'informatique ayant de tout temps été mise à contribution pour développer des applications de gestion, tous les langages proposent les types structurés que vous devez connaître puisque c'est un savoir lié à l'algorithmique.

Les types structurés permettent de représenter une information complexe : une variable d'un type structuré est en fait un container contenant des variables de types de base. Par exemple, un client d'une entreprise est traduit en type structuré *Client*, chaque caractéristique du client (son nom, son prénom, son adresse…) devenant un champ de la structure. Un client physique sera représenté par une variable de type *Client*.

1 B 3. Des variables structurées…

J'insiste sur l'obligation de représentation correcte mais surtout efficace évoquée précédemment. En effet, une représentation correcte est juste une représentation qui *fonctionne*. Par exemple, on peut déclarer des variables indépendantes pour chaque caractéristique de notre client. Dans ce cas, un client réel sera représenté par plusieurs variables.

Cette solution est correcte puisque tout client peut être entièrement représenté par plusieurs variables. En revanche, elle n'est pas efficace car à chaque fois que vous souhaitez manipuler un client, vous devez vous encombrer de plusieurs variables. De plus, il n'est pas possible de faire un tableau de clients dans ces conditions.

C'est la raison pour laquelle les variables structurées sont nécessaires : elles représentent l'aspect correct (les multiples variables précédentes sont toujours présentes, mais en tant que champs de la structure) mais aussi l'aspect efficace puisque tous les champs sont imbriqués dans la structure elle-même. Le client est donc représenté par une seule variable.

Cela fonctionne très bien… mais on peut faire mieux encore !

1 B 4. … aux objets

Résumons-nous : les structures permettent de représenter des concepts du monde réel (le client, la zone de texte, le bouton…), par opposition aux variables simples qui ne représentent que des données.

Nous souhaitons donc manipuler par programmation des concepts évolués : clients, factures, contrôles… Nous cherchons à les voir comme un tout cohérent et non comme un ensemble de données (c'est la raison même des structures). Il est intéressant d'aller plus loin : un concept évolué (un objet du monde réel) n'est pas qu'un ensemble de données de base, de même qu'un humain n'est pas juste un ensemble d'organes.

Ce que j'essaie d'expliquer, c'est qu'une variable de base n'a pas de fonction dans la vie réelle, elle ne *fait* rien : mon âge, mon adresse, mon poids… ce sont des informations, rien d'autre.

Au contraire, une entité est définie par sa fonction dans le système d'information (si l'on distingue le client, c'est que le client sert à quelque chose). Par exemple :

- le client passe une commande, règle une facture, peut déménager…
- une facture peut être émise, payée, annulée…
- un produit peut être commandé, en rupture de stock…
- une zone de texte peut être remplie, vidée, modifiée…
- un bouton peut être cliqué.

Ainsi, ce qui distingue les entités des variables simples, ce sont les fonctions que ces entités réalisent.

C'est à ce stade de nos cogitations que l'on va introduire le concept d'objet. Je vous résume nos réflexions :

♥ Une entité du monde réel est constituée de deux choses indissociables : de l'information
♥ (des données) et des fonctions (des traitements).

Je vais reprendre mon débat du paragraphe précédent avec les solutions correctes ou efficaces. Pour le moment, nous représentons un client ou une zone de texte avec une structure. Les traitements seront des sous-programmes classiques (procédures ou fonctions). Cette solution est correcte.

Oui, juste correcte, car on éclate en deux notre concept du monde réel qui formait pourtant un tout : d'un côté les données, de l'autre les traitements.

Par exemple, si l'on possède un type structuré *Facture*, on aura sans doute envie de savoir si une facture donnée est ou non payée (évidemment, la structure possède les champs adéquats pour pouvoir répondre à cela). On aura donc une fonction de ce type :

```
fonction Payée (f : Facture) : booléen
```

L'idée force est la suivante :

♥ De même que l'on invente la structure pour regrouper des données formant ensemble un
♥ concept cohérent, on invente l'objet pour regrouper données et traitements formant un
♥ ensemble encore plus cohérent.

Voilà, après quatre pages d'introduction, j'ai lancé le grand mot, que je relance d'ailleurs :

♥ Un objet est constitué du regroupement de données et de traitements (procédures et
♥ fonctions) sur ces données pour modéliser une *chose* du monde réel. Les éléments contenus
♥ dans l'objet (ses données et ses traitements) sont ses membres.

J'insiste sur l'objectif de modélisation d'une chose du monde réel, sous-entendu la modélisation d'un concept cohérent. De même que prendre des informations en vrac (le prénom de maman, l'âge du teckel, la durée moyenne de mes nuits…) et les regrouper pour dire *oh, la belle structure* n'a pas de sens, regrouper les données au hasard et y ajouter des traitements qui n'ont rien à y voir ne formera jamais un objet.

Par exemple, ajouter la fonction *payée* précédente aux données définissant un client ne donnera rien d'intelligible.

De même, si modifier le contenu (soit la propriété *Text*) d'une zone de texte a un sens, modifier le contenu d'un bouton n'en a pas puisqu'un bouton n'a pas de contenu (je vous rappelle que *Caption* est une notion différente).

2. Définition des objets

2 A. Introduction

Je le répète de nouveau : un objet, ce sont des données et les traitements associés, l'ensemble décrivant totalement quelque chose du monde réel.

Lorsque vous voulez définir un objet, il vous faut donc des données et des traitements. Comment trouve-t-on tout cela ? Ma foi, ce n'est pas notre problème ici. Disons qu'il faut y réfléchir sérieusement. Définir tous les traitements que peut réaliser une zone de texte n'est pas immédiat. De même, dire qu'une facture peut être payée, c'est évident. Trouver d'autres traitements (et ils existent) est moins simple.

Notre objectif n'est pas de construire de nouveaux objets, mais juste de savoir utiliser (manipuler) ceux que VB nous propose, à savoir les contrôles. C'est pourquoi nous supposons que les données et les traitements sont déjà fournis.

Je possède des données et des traitements que je souhaite associer pour définir un objet. Comment faire ? C'est pour le moment un problème purement syntaxique.

Vous vous attendez sans doute à ce que je vous sorte une nouvelle syntaxe dévolue aux objets. Eh bien, vous avez raison. Enfin, à moitié : nous allons reprendre quasiment celle des structures, en changeant juste le mot de *structure* en *objet*.

2 B. Vocabulaire

L'objet est un concept moderne représentant la pointe en matière de programmation. C'est très efficace, mais n'a pas vraiment d'intérêt pour les programmes que vous pourriez écrire, raison pour laquelle nous n'étudierons pas ensemble comment faire sous VB pour définir des objets.

En revanche, l'application VB est suffisamment importante pour mettre en œuvre des objets. J'ai dit qu'il y avait les contrôles Windows, mais en fait toute l'infrastructure VB est bâtie avec des objets.

Ce que j'attends de vous est la maîtrise des **concepts** de base et du **vocabulaire** objet pour comprendre le principe de fonctionnement des objets que vous manipulerez au travers de VB et qui se résument, je le répète, aux contrôles Windows.

Nous venons de voir l'analogie entre les structures et les objets. Pour identifier correctement les concepts, on ajoute du vocabulaire.

♥ L'équivalent du concept du monde réel est appelé une **classe**.

Attention, une classe est analogue à un type informatique, à savoir que c'est une définition abstraite (c'est l'Homme par rapport à un homme). Par exemple le type *entier* est analogue à la classe *Client* ou à la classe *TextBox* (type correspondant aux zones de texte).

♥ On dit qu'une variable est d'un type donné et qu'un objet est une instance d'une classe
♥ donnée. Les notions *variable* et *type* deviennent *instance* et *classe*.

♥ Ainsi, de même que *i* est une variable de type entier (bref, *i* est un entier), on dira que
♥ l'objet *Cl1* est une instance de la classe *Client* ou que les contrôles *Départ* et *Arrivée* sont
♥ des instances de la classe *TextBox* (ce qui se dit plus simplement *Départ* et *Arrivée* sont des
♥ zones de texte).

♥ Enfin, les données constituant la classe sont appelées des **propriétés** et les traitements sont
♥ des **méthodes**. Les propriétés et les méthodes sont les **membres** de la classe.

Nous retrouvons le terme *propriété* utilisé dans la séquence précédente.

2 C. Syntaxe de déclaration et d'accès aux membres

2 C 1. Déclaration

Si vous avez assimilé ce qui précède, vous vous attendez à déclarer deux choses : la classe et l'objet (de même que l'on déclare la structure et la variable structurée, soit le type et la variable).

Eh bien, c'est exact.

Comme je l'ai dit précédemment, une classe est pour le moment une structure regroupant des données et des traitements. Enfin, habituons-nous à utiliser le vocabulaire correct : la classe est une structure regroupant propriétés et méthodes.

Les propriétés seront déclarées exactement comme les champs dans une structure ; c'est normal puisque les concepts sont équivalents.

Les méthodes (procédures et fonctions) sont un concept nouveau. Il nous faut une nouvelle syntaxe. Elle sera simple : chaque méthode sera définie par son en-tête (la première ligne contenant le nom du sous-programme). Le corps de la méthode (en clair, les instructions entre les *début* et *fin* du sous-programme) sera défini plus loin.

Voici donc la syntaxe de déclaration de la classe (version algorithmique et non VB). Notez l'analogie avec la structure !

```
type
   NomClasse = Classe
      // propriétés de la classe
      variable1 : type
      variable2 : type
      …
      variablen : type
      // méthodes de la classe
      procédure NomProcédure1 (paramètres)
      …
      procédure NomProcéduren (paramètres)
      fonction NomFonction1 (paramètres) : type
      …
      fonction NomFonctionn (paramètres) : type
   fin classe
```

J'ai d'abord défini les données, puis les procédures et enfin les fonctions. En réalité, il n'y a pas plus d'ordre que dans la définition des structures. Par souci de lisibilité, la norme impose de présenter d'abord les données puis les traitements.

Une fois la classe déclarée, c'est comme si elle avait toujours existé. On peut donc déclarer des instances de cette classe.

La déclaration est tout à fait classique :

```
var
   Objet : Classe
```

Ce code déclare un objet *Objet* comme étant une instance de la classe *Classe*. (Il s'agit toujours de syntaxe algorithmique, nous verrons dans la séquence suivante comment déclarer des variables sous VB.)

2 C 2. Accès aux membres

♥ J'ai déjà dit que la syntaxe était la plus proche possible de celle des structures. Pour accéder
♥ aux propriétés et aux méthodes, on reprendra l'opérateur « . » :

- ♥ – *objet.propriété* donne accès à la propriété *propriété* de l'objet *objet* ;
- ♥ – *objet.méthode (paramètres)* appelle (exécute) le sous-programme *méthode*.

Je vous rappelle que l'objet comprend ses données (comme une structure), mais aussi ses traitements, sous-entendu toutes les manipulations que l'on peut faire sur l'objet lui-même.

2 D. Exemple

2 D 1. Partons d'une structure *Client*

Je déclare une structure *Client* avec ses champs :

```
type
   Client = structure
      Nom : chaîne
      Prénom : chaîne
      Adresse : chaîne
      CodePostal : chaîne
      Ville : chaîne
      DatePremièreCommande : date
   fin structure
```

Comme un client peut déménager, on doit posséder un sous-programme permettant de mettre à jour son adresse. Le voici :

```
procédure DéménagerCl (var Cl : Client, NewAdr : chaîne, NewCP : chaîne, NewVil : chaîne)
début
   Cl.Adresse := NewAdr
   Cl.CodePostal := NewCp
   Cl.Ville := NewVil
fin
```

Voici un exemple de code mettant en œuvre la structure et la procédure :

```
var
   Cli : Client
   Adresse, Code, Ville : chaîne

début
   saisir "Entrez la nouvelle adresse", Adresse
   saisir "Entrez le nouveau code postal", Code
   saisir "Entrez la nouvelle ville", Ville
   DéménagerCl (Cli, Adresse, Code, Ville)
fin
```

2 D 2. Passons à la classe *Client*

Je déclare une classe *Client* avec ses propriétés et ses méthodes (j'en profite pour ajouter une méthode renvoyant – c'est donc une fonction – l'ancienneté d'un client) :

```
type
   Client = classe
      // propriétés
      Nom : chaîne
      Prénom : chaîne
      Adresse : chaîne
      CodePostal : chaîne
      Ville : chaîne
      DatePremièreCommande : date
      // méthodes
      procédure déménager (NewAdr : chaîne, NewCode : chaîne, NewVille : chaîne)
      fonction Ancienneté : date  // renvoie DateJour-DatePremièreCommande
      …
   fin classe
```

Cette classe est un peu folklorique : je ne l'ai écrite que pour illustrer la syntaxe. Si vous deviez l'utiliser dans une application, il faudrait la compléter, bref faire un travail d'analyse.

Notez que les deux méthodes n'ont aucun paramètre faisant référence à un objet *Client*. Est-ce normal ? Oui ! En effet, ces méthodes sont les méthodes d'un objet donné (enfin, plus précisément, comme ces méthodes font partie de la classe, chaque instance de la classe possède les méthodes). On appellera donc les méthodes à partir d'un objet qui jouera implicitement le rôle du paramètre : une méthode étant appelée à partir d'un objet, la méthode s'applique à cet objet.

Comme prévu, la définition de la classe ne donne que l'en-tête des méthodes. Il faut écrire le sous-programme lui-même. Le voici :

```
procédure Client.Déménager (NewAdr : chaîne, NewCode : chaîne, NewVille : chaîne)
début
   Adresse := NewAdresse
   CodePostal := NewCode
   Ville := NewVille
fin
```

Il y a trois changements par rapport à la procédure *DéménagerCl* du paragraphe 2.D.1 :

- le nom de la procédure est préfixé par « Client. ». C'est obligatoire pour préciser que l'on définit la méthode *déménager* de la classe *Client* et non pas celle, par exemple, de la classe *Fournisseur* ;
- aucun paramètre ne représente le client. Nous venons d'évoquer ce point : comme on appelle la méthode à partir d'une instance de *Client*, elle s'appliquera automatiquement à cette instance ;
- dans les membres gauches des trois instructions d'affectation, on utilise directement le nom du champ (oups, pardon, nous manipulons des objets donc il faut employer le terme exact qui est *propriété*). C'est exactement le même raisonnement que pour le paramètre qui semblait avoir disparu. La méthode est appelée à partir d'un objet. Comme elle fait partie de ce dernier, nous sommes en quelque sorte à l'intérieur de l'objet lorsque nous exécutons la méthode. Les propriétés sont donc directement accessibles.

Voici un exemple de code mettant en œuvre l'objet et la méthode :

```
var
   Cli : Client
   Adresse, Code, Ville : chaîne

début
   saisir "Entrez la nouvelle adresse", Adresse
   saisir "Entrez le nouveau code postal", Code
   saisir "Entrez la nouvelle ville", Ville
   Cli.Déménager (Adresse, Code, Ville)
fin
```

2 E. Ne nous emballons pas !

Attention, je vous rappelle que nous n'allons pas faire un cours de programmation objet ici. Mon objectif n'est pas de vous apprendre à concevoir des objets, mais uniquement à manipuler ceux qui vous sont proposés par VB.

Cela passe évidemment par la connaissance de :

- ce qu'est un objet (des données et des traitements réunis de façon cohérente) ;
- le vocabulaire officiel (instance et classe, propriétés, méthodes et membres) ;
- la syntaxe permettant d'accéder à un membre (l'opérateur « . »).

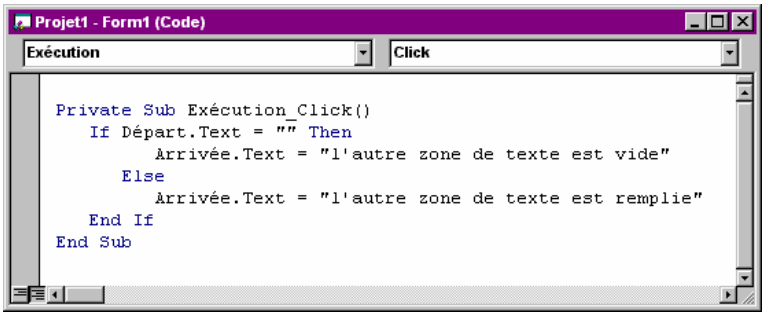
Les concepts qui font que la programmation objet est intéressante (l'héritage, le polymorphisme…) ne sont pas abordés. Du coup, pour vous, l'objet est uniquement une construction syntaxique différente des structures classiques : au lieu de faire des procédures séparées, elles sont incluses dans l'objet. Il n'y a rien de plus à savoir.

Ce que nous venons de voir peut vous sembler relativement théorique. Pour vous prouver que cela nous permet néanmoins de comprendre parfaitement le code que VB avait généré tout seul dans la séquence précédente, nous allons l'étudier de nouveau.

2 F. Retour sur la séquence précédente

Dans le paragraphe 2D2 de la séquence 3, nous avons analysé le code lié à la feuille.

Nous pouvons maintenant comprendre de façon plus théorique le code écrit. Reprenons le code événementiel que nous avons écrit :



Voici les conséquences du modèle objet :

- la feuille *Form1* est un objet ;
- les différents contrôles qu'elle contient (*Départ*, *Arrivée* et *Exécution*) sont eux-mêmes des objets et sont des propriétés de la feuille, cela traduisant bien que la feuille contient les contrôles ;
- *Text* est une propriété de tout objet zone de texte (donc de *Départ* et *Arrivée*) ;
- plus important, *Click* est une méthode du contrôle *Exécution*.

Quel enseignement tirer de cela ? Que les procédures événementielles (le code déclenché par la survenue d'un événement) sont des méthodes.

Finalement, c'est logique : le code déclenché par l'événement *click* sur un bouton est un code exécuté par ce bouton : on dira que le bouton réagit (par l'exécution de code) à l'événement qui vient de lui arriver.

Cela vous semble moyennement clair ? Eh bien, attaquez le paragraphe suivant qui parle des événements. Et alors, cela semblera clair, limpide, simple, évident…

3. Les événements

3 A. Généralités

Nous allons reprendre de façon plus formelle ce que nous avions commencé dans la séquence précédente.

Tout système d'exploitation graphique (Windows et autres) propose à l'utilisateur deux outils pour fournir de l'information au système : le clavier et la souris. Voici un bref descriptif de leur rôle :

- le clavier est l'élément le plus naturel pour saisir du texte. Si, au lieu d'utiliser le clavier, vous deviez cliquer à la souris les différents caractères que vous voulez taper pour créer un document sous Word, vous seriez bien malheureux !
- la souris n'est pas faite pour taper du texte. Elle est en revanche idéale pour sélectionner des commandes (clic sur les barres d'outils et de menus), pour lancer des applications (clics sur des icones…).

Un système graphique est :

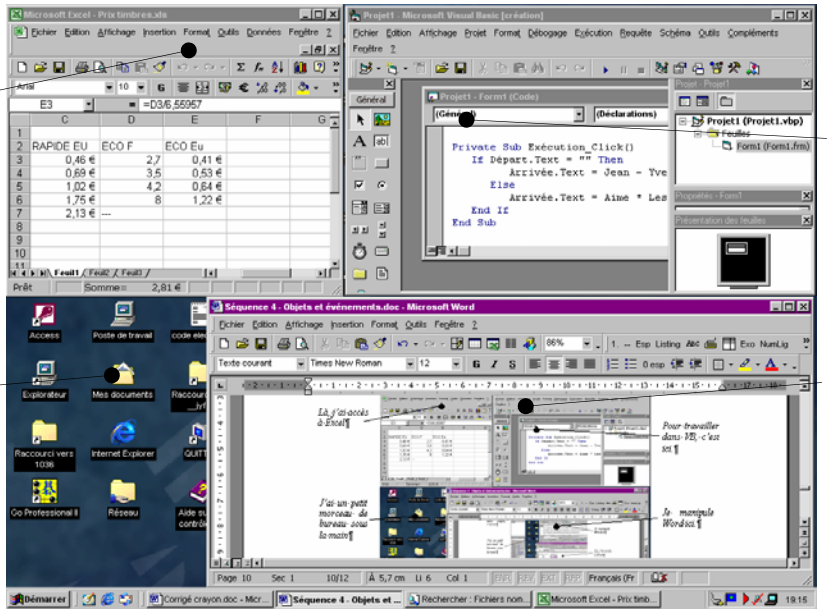
- intéressant pour son ergonomie (la simplicité et le confort de la manipulation de ses différents composants) ;

– nécessaire dans un monde multi-tâches où différentes fenêtres auxquelles on accède en cliquant dedans exécutent différents programmes.

Les premiers systèmes, que vous avez peut-être connus (par exemple, le DOS) n'étaient pas graphiques (ni multi-tâche d'ailleurs). Pour communiquer avec le système, il fallait taper une commande au clavier puis faire *Enter*. La commande était alors traitée.

Un système graphique vous offre en permanence de nombreux points d'interaction. Par exemple, une fois Windows démarré, vous pouvez cliquer dans le menu *Démarrer*, mais aussi sur n'importe quel élément présent sur le bureau.

Allons plus loin : si vous lancez VB, Word et Excel et réorganisez les fenêtres pour toutes les avoir à l'écran, vous pouvez agir partout :



La navigation entre les différentes applications est permise par la fonction multi-tâche du système d'exploitation.

3 B. Rôle du système d'exploitation

Comment une application sait-elle que vous avez cliqué sur un de ses bouton ? Par exemple, dans la copie d'écran ci-dessus, vous avez trois icones *Enregistrer* (la petite disquette en troisième position dans chaque barre d'outils). Si vous cliquez sur l'un des trois, comment l'application concernée le saura-t-elle ?

Bon, vous allez me dire que si, au sein d'une foule, quelqu'un vous touche, vous le saurez sans que l'on ait besoin de vous prévenir. Cela est rendu possible parce que votre système nerveux, votre corps et votre cerveau forment un tout.

Dans le cadre de la programmation événementielle, c'est différent : ces trois constituants sont éclatés. Reprenons-les un par un. L'équivalent de :

– votre corps (la *chose* que l'on touche), c'est l'interface graphique de l'application, à savoir ses différentes feuilles ;
– votre système nerveux (ce qui achemine l'information au centre de commande), c'est le système d'exploitation ;
– votre cerveau (le centre de commande qui décide quoi faire en fonction des informations qui arrivent du système nerveux), c'est le code de l'application.

Ainsi, lorsque vous utilisez une application, vous avez l'impression de communiquer avec elle : vous activez une commande (en cliquant sur un bouton ou un menu) et l'application répond en exécutant le traitement souhaité.

En fait, c'est une illusion. La communication se fait par le truchement du système d'exploitation : c'est ce dernier qui va détecter que l'utilisateur a cliqué sur un contrôle d'une application et qui va envoyer un message à l'application pour lui dire « l'événement *X* vient de se produire sur le contrôle *Y* »[1] (je résume un peu).

Vous devez retenir deux choses de cette explication :

- d'une part, nous sommes dans un cadre de travail action/réaction : le système réagit à des événements qui se produisent. La plupart du temps, c'est l'utilisateur qui déclenche les événements avec la souris ou le clavier. La réaction du système, c'est l'exécution de code ;
- d'autre part, c'est le système d'exploitation qui perçoit chaque événement et qui le communique à l'application concernée.

Le fait que les événements transitent par le système d'exploitation explique pourquoi vous ne pouvez pas ajouter vos propres événements dans une application[2]. Supposons que vous vouliez créer l'événement *TripleClic* qui se déclencherait lorsqu'un utilisateur clique trois fois rapidement sur un contrôle. Comment faire ? C'est impossible au sein de votre application car pour que le programme soit averti de la survenue d'un triple clic, il faudrait que le système d'exploitation l'en avertisse. Or, ce dernier ne connaît pas la notion de triple clic. Il est donc incapable de détecter sa survenue[3].

3 C. Soyons précis

3 C 1. Avec le vocabulaire

Pour que le programme sache quoi faire en réponse à un événement, il faut qu'il sache :

- quel est cet événement (clic, mise à jour, survol souris…) ;
- quel est le contrôle concerné (sur quel contrôle on vient de cliquer, ou quel contrôle vient d'être mis à jour ou survolé par le curseur…).

Du coup, l'événement sera toujours associé à un contrôle. Partant du principe qu'un événement seul (on a cliqué dans l'application) n'est pas informatif, ma définition de l'événement va inclure le contrôle.

Je vous propose la définition suivante :

♥ Un événement est quelque chose de répertorié arrivant à un contrôle.

Deux remarques :

- les termes *quelque chose* ne font pas très professionnels, mais quoi utiliser d'autre ? Le terme événement est tellement explicite…
- le mot *répertorié* signifie que seuls existent pour le système (et donc seuls sont pris en compte) les événements qui sont définis dedans. La seule chose à faire, c'est donc d'espérer que les développeurs de Windows aient pensé à tous les événements dont nous pourrions avoir besoin.

[1] Par exemple « l'événement *clic* vient de se produire sur le contrôle *OK* » (*OK* est un bouton).
[2] Attention, je parle bien d'événement, pas du code lié à un événement (nous allons aborder cela).
[3] En fait, c'est encore pire que cela : comme il connaît l'événement double clic, le système préviendra l'application que le double clic a eu lieu dès que vous aurez cliqué deux fois sur les trois que compte votre triple clic. Ce dernier sera donc détecté comme deux événements : un double clic suivi d'un clic.

Continuons dans le vocabulaire.

♥ Une procédure événementielle est une procédure associée à un événement. Elle est
♥ exécutée lorsque l'événement a lieu.

3 C 2. Avec les concepts

Je tiens à faire une petite précision. Nous avions vu, dans le paragraphe 2B de la séquence 3, que cliquer sur un bouton lorsque aucun code n'était attaché à l'événement clic de ce bouton ne servait à rien : il ne passait rien.

Il est bon de comprendre que l'événement a lieu. Il est donc perçu par Windows qui communique l'information à l'application. Comme l'application n'a aucun code à exécuter, elle ne réagit pas.

L'événement a donc toujours lieu. En revanche, la procédure événementielle peut être présente ou non.

3 C 3. La programmation événementielle

Faire un programme événementiel revient donc à écrire le bon code dans les bons événements.

3 D. Lien entre les objets et les événements

Nous avons dit qu'un objet était *quelque chose* (plus ou moins une structure) contenant non seulement des propriétés (données) mais aussi des méthodes (traitements).

Les propriétés et méthodes concernent évidemment l'objet. Eh bien, cela nous permet de fusionner les deux concepts d'objets et d'événements.

En première approximation, on dira que chaque événement est implanté comme une méthode de l'objet concerné. Ainsi, si les boutons possèdent une méthode *Click*, c'est parce que l'événement éponyme a un sens pour eux. Si vous ne définissez rien pour *Click*, la méthode est vide. Lorsque l'événement se produit, la méthode est exécutée, mais, comme elle est vide, il ne se passe rien[1].

Voilà. Avec ce savoir, nous pouvons travailler efficacement dans VB. Le moment est donc venu de découvrir les instructions du langage basic.

[1] En réalité, c'est un peu plus subtil (si vous ne saisissez pas ce qui suit, cela n'a aucune importance). *Click* n'est pas une méthode mais une propriété d'un type un peu spécial : elle ne contient pas un entier ou une chaîne, mais une procédure (enfin, l'adresse en mémoire d'une procédure). Et quelle procédure ? Eh bien, la procédure événementielle que vous avez éventuellement écrite. Donc soit il y a une adresse (si vous avez écrit la procédure) soit il y a la constante *NIL* représentant conventionnellement l'absence d'adresse.
Ainsi, lorsque l'événement *Click* d'un bouton se produit, l'application va voir si sa propriété *Click* possède une adresse. S'il n'y en a pas (donc *Click* vaut *NIL*), il ne se passe rien. Sinon, la procédure stockée à l'adresse trouvée est exécutée.

Séquence 5

VB : présentation du langage

Il est temps de découvrir le langage VB pour pouvoir écrire de vrais programmes !

Contenu

Description du langage VB : instructions, types et sous-programmes.

L'interface utilisateur de VB.

Capacités attendues

Savoir traduire un algorithme sous VB.

1. Introduction

1 A. Approche de VB

Je pourrais vous écrire un livre sur ce langage, mais ce n'est pas notre propos. Je me contenterai donc de vous en montrer juste assez pour que vous puissiez vous en servir. Cela tiendra en trois grandes parties :

- la traduction sous VB des différents concepts de l'algorithmique ;
- les erreurs classiques d'un débutant VB ;
- les modules.

VB est un langage de développement événementiel au même titre que Delphi, Visual C++… Vous avez donc accès aux feuilles et à tous les contrôles visuels de Windows. Vous pouvez également travailler en mode console (texte) sans utiliser le mode graphique.

1 B. L'autonomie avec VB

Lorsque vous allez lire cette séquence, vous vous direz avec raison qu'il n'y a pas de difficulté majeure dans VB. Cela dit, je suis prêt à parier que vos premiers programmes auront de très nombreuses erreurs de compilation. C'est normal puisque ce type d'erreur révèle un problème de syntaxe. Et la syntaxe, c'est justement ce dont on ne se préoccupait pas en algorithmique.

Si j'étais avec vous, je regarderais un quart de seconde votre écran pour vous aider à identifier votre erreur et je vous dirais d'un air désabusé « manque un *End If* », « manque *End If* » voire « *End If* ! »[1].

Seulement voilà, si vous lisez ce cours, c'est que je ne suis pas derrière vous. Autonomie, débrouillardise et autonomie seront vos mots d'ordre. N'hésitez pas à consulter l'aide en ligne du langage : mettez votre curseur texte sur un mot du langage (pour cela, déplacez-le avec les flèches ou cliquez dessus), puis faites *F1*. Vous obtenez l'aide relative au mot ou à l'instruction.

[1] Où toute autre phrase laconique expliquant la grosse erreur de syntaxe faite.

2. Premier contact : les principes de base

2 A. Encore une mise en garde

Je l'ai dit, je le répète : ceci n'est pas un livre complet sur VB vous apprenant tout dans les moindres détails. Vous allez devoir vous empoigner pour travailler sur VB et en découvrir toutes les fonctionnalités.

Vous pouvez envisager l'achat d'un livre de référence puisque cette séquence n'en tient pas lieu. Cela dit, est-ce nécessaire ? Je n'en suis pas convaincu. Si vous souhaitez approfondir VB dans ses concepts les plus poussés (accès aux bases de données, programmation objet[1]), alors vous devrez vous tourner vers la littérature du commerce puisque le CNED ne propose pas encore cette formation. Cela dit, attaquer ces concepts pointus n'a de sens que quand vous maîtrisez parfaitement les bases contenues ici. Ce n'est qu'une fois ce cours bien étudié, parfaitement assimilé et après l'écriture de nombreux programmes pour être à l'aise dans le langage que vous pourrez envisager d'aller plus loin.

Je vous conseille de nouveau d'utiliser chaque fois que possible l'aide en ligne.

2 B. Organisation d'un programme VB

2 B 1. Introduction

VB est un langage de développement événementiel mettant en œuvre l'ergonomie Windows. Vous avez accès à tout l'environnement graphique : fenêtres, contrôles, souris et événements.

Vous pouvez également nier une dizaine d'années de progrès en informatique et développer une application console en mode texte. Vous obtenez alors l'équivalent de ce que faisait le vieux langage Qbasic. Le mode console correspond à l'algorithmique puisque cette dernière ne gère pas les événements.

Bien entendu, nous travaillerons en mode événementiel, soit en utilisant l'interface graphique de VB, comme nous avons commencé à le faire dans la séquence 3.

2 B 2. Application événementielle

Cas général

Une application événementielle est à la norme Windows : les différents traitements sont lancés par l'utilisateur par le biais du déclenchement d'événements (le plus employé étant le clic souris sur un bouton). Des événements internes à l'application, donc non directement déclenchés par l'utilisateur, permettent aussi d'exécuter du code.

Le code est toujours écrit avec un langage procédural classique (le Pascal, le Basic, le C…). Cela signifie que le code associé aux événements l'est sous la forme d'une procédure. Ainsi, lorsque l'on dit que le clic sur le bouton *Valider* déclenche l'enregistrement des données, cela veut dire que le développeur a associé à l'événement *clic* du bouton *Valider* la procédure réalisant l'enregistrement. Vous conservez la faculté d'écrire des sous-programmes (procédures et fonctions) utilisées par les procédures événementielles.

[1] Sachant que VB (dans sa version 6 tout du moins) ne propose qu'une partie du formalisme objet. Il est donc impropre de parler de langage objet.

Bref, passer d'une application algorithmique classique à une application graphique événementielle, c'est changer l'habillage et l'organisation, soit l'ergonomie. Le code reste le même.

L'interface est réalisée dans des feuilles utilisant pour les saisies et les affichages les contrôles Windows classiques (zone de texte, de liste, case à cocher…).

Tous les langages de programmation événementiels (Delphi, VB, Visual C++…) fonctionnent ainsi. Il faut distinguer deux choses : l'aspect événementiel (les feuilles, contrôles, événements) et le langage de programmation sous-jacent (le Pascal pour Delphi, le Basic pour VB, le C++ pour Visual C++…).

J'insiste : tous les contrôles Windows et leurs événements associés se retrouvent dans ces langages. Mieux encore, ils sont toujours accessibles de la même façon, sous la forme d'objets (voir la séquence précédente). Ainsi, activer un contrôle, changer sa couleur ou sa taille[1] se fera exactement de la même façon sous Access, VB, Delphi…

L'organisation d'une application événementielle est toujours la même : l'aspect graphique (l'interface) est constitué d'un ensemble de feuilles contenant des contrôles. Comme le code est déclenché par la survenue d'un événement associé à un contrôle ou à une feuille (on clique sur tel bouton, on ferme telle fenêtre), on associe à chaque feuille tout le code qui le concerne.

Finalement,

♥ Une feuille = des contrôles + le code associé aux événements de la feuille et de ses contrôles

♥ Un programme événementiel = un ensemble de feuilles

Cas de VB

Ce qui précède est commun à tous les langages de programmation événementiels[2]. Cependant, dès que la notion de programmation pointe sa petite tête, les caractéristiques du langage hôte interviennent.

C'est le cas dans l'équation suivante :

Une feuille = des contrôles + le code associé aux événements de la feuille et de ses contrôles

Les différentes instructions seront évoquées dans la suite de cette séquence. Je souhaite vous parler ici de l'organisation des programmes : comment assembler le code à la feuille ? Peut-on écrire du code indépendant de toute feuille ? Nous verrons tout cela dans le paragraphe 2C.

2 B 3. Application console (que nous n'utiliserons pas beaucoup)

C'est la programmation *à la papa*. Enfin, comme l'informatique va très vite, le papa en question est jeune. Je me réfère à la programmation sous Dos, donc avant Windows 95. Cela nous ramène aux années 80 jusqu'en 1995.

On parle d'application console (ou application en mode texte) car on travaillait dans un environnement texte et non graphique (en vrac : pas de souris ni de fenêtre, peu ou pas de couleur…). L'ergonomie était limitée. Mais bon, on ne connaissait rien d'autre donc cela ne nous déplaisait pas. Évidemment, si l'on devait maintenant revenir à cette programmation, on se sentirait à l'étroit.

[1] Changer la taille, la couleur ou l'emplacement d'un contrôle lors de l'exécution de l'application est aisé à réaliser mais viole toutes les règles d'ergonomie ! Ne vous y amusez pas.

[2] *événementiels* ou *événementielle* ? Pour moi, c'est un langage permettant de mettre en œuvre la programmation événementielle. C'est donc un langage événementiel de programmation, soit un langage de programmation événementiel.

Une application console est un programme sans interface graphique.

Vous connaissez le mode de programmation d'une application console puisque cela correspond exactement à l'algorithmique traditionnelle[1] : un programme est une suite d'instructions dont l'exécution commence à la première, puis passe à la deuxième, la troisième… jusqu'à la dernière. L'interactivité avec l'utilisateur est limitée : le programme affiche des données et l'utilisateur peut en saisir.

C'est ce type de programmation ringarde que nous allons mettre en œuvre avec les fichiers le temps de découvrir ces derniers. Pourquoi ? Parce que la programmation événementielle n'est qu'un habillage visuel (concernant l'ergonomie) d'un programme console classique. Ainsi, même si l'événementiel apporte ses propres difficultés, l'essentiel du travail lors de l'écriture d'un programme reste l'aspect purement algorithmique.

La programmation console sous VB exploite la notion de module… voir ci-dessous.

2 C. Les modules

2 C 1. Définition

Les modules sont utiles pour regrouper du code indépendamment de toute feuille. Vous vous demandez sans doute à quoi cela peut servir puisque notre but est justement d'écrire du code lié aux contrôles des feuilles.

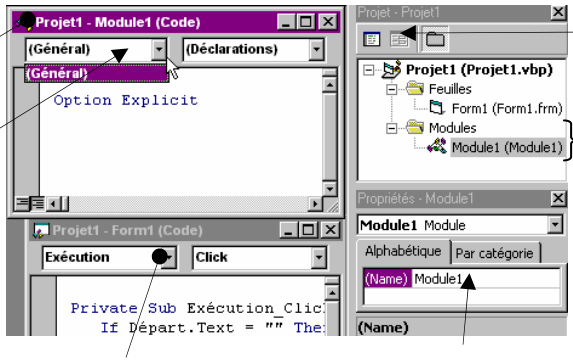
Par exemple, si vous développez du code pour gérer des comptes bancaires (un type *Compte* et des sous-programmes pour ouvrir, approvisionner et fermer un compte), il est intéressant de tout regrouper dans un module qui pourra être utilisé dans n'importe quel programme manipulant des comptes bancaires.

En fait, vous aurez compris que le module permet de regrouper du code utilisé dans plusieurs feuilles (voire dans plusieurs projets).

Comment utiliser un module dans une application ? Il suffit de l'ajouter dans le projet grâce à la commande *Projet/Ajouter un module*. Vous aurez deux onglets permettant d'en créer un nouveau ou de faire référence à un module déjà existant.

Pour fixer les idées, j'ai ajouté un nouveau module à mon application. Voici ce qui se passe alors :

Voici la fenêtre du module que nous venons d'ajouter. On voit bien que c'est un module car la zone de liste des objets ne contient rien, ni contrôle ni feuille. (Notez Option Explicit *suite à notre précédent paramétrage !)*



Le bouton Objet *est désactivé puisqu'un module n'est associé à aucune feuille.*

Le projet s'est doté d'une rubrique Modules *dans laquelle trône notre nouveau module.*

Le code associé à notre feuille est toujours là.

Le module est un objet ne possédant qu'une propriété, son nom.

[1] Dit autrement, l'algorithmique ne s'est pas encore adaptée à l'événementiel.

Il faut bien comprendre que le code associé à une feuille n'est pas différent de celui que l'on trouve dans un module (enfin, procédures événementielles mises à part).

En fait, le code du module est stocké dans un fichier séparé d'extension *.bas* (pour BASic), tandis que le code lié à la feuille est directement stocké dans le fichier de cette dernière à la suite de la description de ses contrôles.
Pour preuve, voici le fichier *Form1.frm* décrivant notre feuille :

```
01 VERSION 5.00
02 Begin VB.Form Form1
03    Caption         =   "Form1"
04    ClientHeight    =   1860
05    ClientLeft      =   60
06    ClientTop       =   345
07    ClientWidth     =   4725
08    LinkTopic       =   "Form1"
09    ScaleHeight     =   1860
10    ScaleWidth      =   4725
11    StartUpPosition =   3  'Windows Default
12    Begin VB.TextBox Arrivée
13       Height          =   315
14       Left            =   1695
15       TabIndex        =   2
16       Top             =   930
17       Width           =   2610
18    End
19    Begin VB.CommandButton Exécution
20       Caption         =   "Exécuter"
21       Height          =   345
22       Left            =   375
23       TabIndex        =   1
24       Top             =   915
25       Width           =   1110
26    End
27    Begin VB.TextBox Départ
28       Height          =   300
29       Left            =   345
30       TabIndex        =   0
31       Top             =   240
32       Width           =   1740
33    End
34 End
35 Attribute VB_Name = "Form1"
36 Attribute VB_GlobalNameSpace = False
37 Attribute VB_Creatable = False
38 Attribute VB_PredeclaredId = True
39 Attribute VB_Exposed = False
40 Option Explicit
41
42
43 Private Sub Exécution_Click()
44    If Départ.Text = "" Then
45          Arrivée.Text = "l'autre zone de texte est vide"
46       Else
47          Arrivée.Text = "l'autre zone de texte est remplie"
48    End If
49 End Sub
```

Expliquons un peu ces lignes :

- lignes 2 à 34, nous décrivons l'objet *Form1*, constitué de :
  - ses propriétés classiques (lignes 3 à 11) ;
  - un objet zone de texte appelé *Arrivée* (lignes 12 à 18) ;
  - un objet bouton appelé *Exécution* (lignes 19 à 26) ;
  - un objet zone de texte appelé *Départ* (lignes 27 à 33) ;
- lignes 35 à 39, il y a des choses pas très claires ;
- ligne 40, nous retrouvons quelque chose de connu (et important !) ;

– lignes 43 à 49, il y a notre procédure événementielle liée au bouton *Exécution* défini plus haut.

Cette démonstration n'avait d'autre objet que vous prouver que le code lié à une feuille est indissociable de la feuille elle-même puisque les deux constituent un seul fichier. Attention, il n'est pas question de modifier directement ce fichier, il n'est pas fait pour cela. Pour travailler sur la feuille et son code, ouvrez le fichier sous VB.

2 C 2. Rôle

VB exploite la notion de *module*, que vous fassiez de la programmation événementielle ou console.

Un module, c'est quoi ? C'est un concept simple et intuitif que l'on retrouve sous une forme plus ou moins proche dans tous les langages de développement (fichiers inclus, librairies, unités…).

Je tourne autour du pot… Un module est tout simplement un ensemble de types, de variables et de sous-programmes (procédures et fonctions). En fait, c'est l'équivalent d'un programme, à la grosse différence près que le module ne possède pas forcément de programme principal. (Concrètement, un module sera un fichier source sur votre disque, tout comme un programme.)

L'intérêt d'un module, c'est d'être utilisé par un programme. Cela permet la réutilisabilité du code : si vous vous spécialisez dans la conception d'activités bancaires, vous aurez toujours besoin du type *Compte* et des sous-programmes *ouverture*, *fermeture*, *crédit* et *débit*.

La solution la moins efficace consiste à réécrire (ou ajouter par copier/coller) ces type et sous-programmes dans chaque programme que vous allez écrire. Cette solution n'est pas terrible car si vous souhaitez modifier le type *Compte* ou la procédure *ouverture*, vous devrez modifier chacun des programmes déjà écrits.

Le bonne idée, c'est de mettre les sous-programmes et le type *Compte* dans un module appelé *banque* (ou n'importe quel autre nom intuitif). Chacun des projets en ayant besoin fera référence à ce module. C'est alors comme si son contenu était inclus dans le programme.

Vous commencez sans doute à mieux cerner l'intérêt des modules : vous pouvez vous écrire (ou acheter) des modules variés : un gérant les graphiques, un autre les sons, la souris, les calculs scientifiques et financiers… En fonction des programmes que vous écrirez, vous utiliserez un ou plusieurs de ces modules. Tout ce passe comme si vous vous dotiez d'extensions de VB.

2 C 3. Syntaxe d'un module

Un module contient deux parties :

– la partie déclaration, où l'on trouve *Option Explicit* puis tous les types et variables globaux au module ;
– tout le reste, c'est-à-dire les procédures et les fonctions du module.

Je vous rappelle que ce qui vaut pour le module vaut également pour le code lié aux feuilles puisque la seule différence entre eux est leur localisation physique (dans un fichier indépendant ou dans le fichier de déclaration de la feuille).

3. Généralités sur un programme VB (application console)

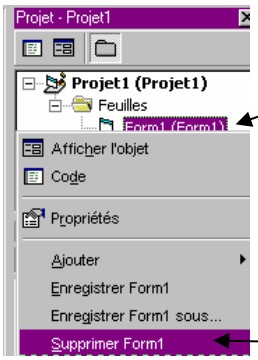
Dans cette séquence, nous allons découvrir le langage VB. C'est pourquoi nous ne travaillerons qu'en mode console pour ne pas être perturbés par l'aspect visuel. Dès la séquence suivante, la programmation événementielle reprendra ses droits.

La syntaxe algorithmique est traditionnellement un mélange des syntaxes du Pascal et du VB. C'est dire que vous ne serez pas trop perdu !

Une application console ne contient pas de feuille. Il faut donc au moins un module contenant le code qui sera exécuté !

Pour créer une application console depuis VB, c'est simple :

- vous créez un nouveau projet ;
- vous supprimez la feuille créée automatiquement. Comment ? Dans la fenêtre *Projet*, vous faites un clic droit sur la feuille puis choisissez *Supprimer Form1*. Visuellement :

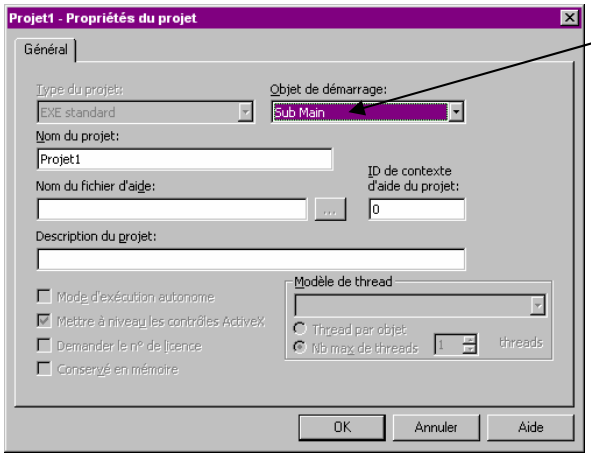


Clic droit ici

On sélectionne cette commande

Il faut ensuite ajouter un module (commande *Projet/Ajouter un module*). Voilà… le tour est joué, reste à écrire le code.

Il est temps pour vous de découvrir la boîte de dialogue obtenue par la commande *Projet/Propriété de Projet1*… (c'est la dernière commande du menu) :



C'est cette zone de liste qui m'intéresse. Il y est dit que l'objet de démarrage est Sub Main. Sub *signifie* procédure, Main principal. Sub Main, *c'est le programme principal. Votre module devra donc contenir une procédure appelée* Sub Main. *C'est elle qui sera exécutée quand vous lancerez l'application. Ce sera le cas pour tout programme console.*

Rechargez notre précédente application de la séquence 3 et retournez voir cette boîte de dialogue. Vous verrez que l'objet de démarrage était la feuille Form1.
Cette zone de liste indique donc ce qui doit être lancé au démarrage de l'application.

4. Les types et les variables

4 A. Les chaînes de caractères

Sous VB, les chaînes de caractères sont encadrées par des guillemets et non des apostrophes : "*Hello !"* et non ~~*'Hello !'*~~.

4 B. Types

4 B 1. Types intégrés

Voici les mots-clé définissant les différents types :

ALGORITHMIQUE	VB
entier	integer
réel	single
booléen	boolean
chaîne	string
date	date

Vous remarquerez que l'on se contente de traduire en anglais.

4 B 2. Types structurés

On utilise beaucoup les types personnalisés pour créer des types structurés (également appelés types enregistrement). Rappelons brièvement ce que sont ces types.

Supposons que je veuille manipuler des clients (enfin, des variables modélisant des clients) dans un programme. J'ai établi qu'un client, c'est :

- un numéro ;
- un nom ;
- un prénom ;
- une adresse ;
- un téléphone ;
- la date à laquelle il est devenu client.

Si je veux manipuler un client appelé *Client1*, je n'ai pas d'autre choix que de déclarer une variable par propriété énumérée ci-dessus. Cela donne :

```
var
   NumClient_Client1 : entier
   NomClient_Client1 : chaîne
   PrénomClient_Client1 : chaîne
   AdrClient_Client1 : chaîne
   TélClient_Client1 : chaîne
   DateClient_Client1 : date
```

Est-ce satisfaisant ? Non, pour au moins quatre raisons :

- pour manipuler *Client1* dans un sous-programme, je dois passer pas moins de six variables en paramètre ;
- si je veux déclarer un nouveau client, je dois de nouveau déclarer six variables (*NumClient_Client2*…) ;
- impossible de stocker mes clients dans un tableau car ce dernier ne contient que des éléments de même type ;

– enfin, la modélisation de la réalité n'est pas très efficace. En effet, dans cette réalité, un client n'est pas juste la réunion des six caractéristiques que j'ai énumérées : un client, c'est un client.

Je souhaiterais pouvoir définir un type *Client* contenant toutes les caractéristiques qui m'intéressent. Je pourrais alors déclarer mes variables client ainsi :

```
var
   Client1, Client2, Client3, Client4 : Client
```

Pour faire cela, on introduit le concept de type structuré, à savoir type contenant d'autres variables. En algorithmique, on le définit ainsi :

```
type
   nom = structure
            variable1 : type1
            variable2 : type2
            ...
            variablen : typen
         fin structure
```

Les variables constituant le type sont appelées des champs. Toute variable d'un type structuré possède tous les champs définis dans le type. L'ordre des champs dans la structure n'a aucune importance.

Notre type *Client* sera défini ainsi :

```
type
   Client = structure
               Num : entier
               Nom : chaîne
               Prénom : chaîne
               Adr : chaîne
               Tél : chaîne
               DateClient : date
            fin-structure
```

Il devient inutile de suffixer chaque champ par *_Client* car nous sommes dans la structure. Pour accéder au champ *Nom* de la variable *Cl1* (de type *Client*), j'écrirai *Cl1.Nom*.

Traduire cela sous VB ne sera pas très difficile : allez dans l'aide de VB et cherchez le mot clé *type*. Cela vous donnera la syntaxe à utiliser.

Exercice 10

Définissez le type *Client* ci-dessus, cette fois avec la syntaxe VB. Ne le faites pas encore dans un programme, écrivez-le à la main.

Exercice 11

Nous allons tester cela sous VB. Nous restons en mode console. Faites-en une nouvelle comme nous l'avons vu ci-dessus puis déclarez le type *Client* après *Option Explicit*.

Exécutez votre programme. Si tout est correct, il ne se passera rien puisqu'il n'y a pas de code. Si vous avez une erreur quelconque, le compilateur vous la signalera. Essayez alors de la corriger avant d'aller lire la correction.

4 C. Variables

Une variable est déclarée en donnant son nom et son type, le mot-clé *as* séparant les deux. Toutes les déclarations sont précédées du mot-clé *dim* (indiquant justement que l'on déclare des variables).

Les déclarations de variables sont faites après celles des types (s'il y en a) et avant tout sous-programme (si l'on déclare des variables globales).

✎Exercice 12✎

Déclarez *i* et *j* (deux entiers) et *Cl* (un client du type précédent) dans notre programme. Exécutez-le pour vérifier que la syntaxe est correcte.

Lorsque plusieurs déclarations de variable se suivent, on peut les déclarer sur une ligne avec un seul mot-clé *dim* et en les séparant par des virgules.

En reprenant l'exercice précédent, on obtient :

```
dim i as integer, j as integer, Cl as Client
```

Attention à l'erreur classique sous VB : la déclaration de plusieurs variable d'un coup avec le même type n'est pas possible comme cela se fait sous Delphi et C.

Ainsi, l'écriture suivante est incorrecte :

```
dim i, j as integer
```

En effet, vous croyez déclarer deux entiers *i* et *j* ? Eh bien non ! Vous déclarez une variable *j* qui est entière et une variable *i* sans type (donc de type *variant*).

Je vous conseille de déclarer toujours vos types avant les variables. Cela ne changera rien au fonctionnement du code mais il est plus logique de déclarer les choses avant de les utiliser.

La règle est donc de déclarer tous les types puis les variables.

4 D. Accès aux variables structurées

Pour accéder à un champ d'une variable structurée, on utilise la syntaxe algorithmique :

structure.champ

Supposons que nous voulions vider la structure *Client1* de type *Client*, soit mettre des chaînes vides[1] dans tous les champs *chaîne* et 0 dans tous les champs numériques et dates.

Nous devrions écrire :

```
Client1.Num = 0
Client1.Nom = ""
Client1.Prénom = ""
Client1.Adr = ""
Client1.Tél = ""
Client1.DateClient = 0
```

Le côté fastidieux ne vous échappe pas : il faut écrire « Client1. » de nombreuses fois. Si la structure était incluse dans une autre et/ou dans un tableau, on aurait une expression encore plus longue à répéter. Non seulement c'est long à écrire, mais ce n'est pas très lisible.

Tous les langages possèdent une instruction permettant de factoriser un nom de structure. Sous VB, c'est *with*.

La syntaxe est la suivante :

```
With NomStructure
   // les champs de la structure NomStructure
   // n'ont pas à être préfixés par « NomStructure. »
   // mais seulement par le « . » rappelant qu'ils viennent d'une structure
End With
```

(Voir l'aide de VB pour plus d'explications.)

Voici l'utilisation du *with* dans notre initialisation du client :

[1] Une chaîne s'encadre avec le caractère guillemet (« " », touche 4 du clavier). Comme la chaîne est vide, elle commence par un guillemet, puis se termine immédiatement, d'où un autre guillemet.

```
With Client1
   .Num = 0
   .Nom = ""
   .Prénom = ""
   .Adr = ""
   .Tél = ""
   .DateClient = 0
End With
```

4 E. Tableaux

4 E 1. Déclarations

La syntaxe du VB est intéressante car vous pouvez spécifier vos indices de trois façons :

- en donnant l'indice de début et celui de fin ;
- en donnant l'indice de fin, le premier ayant implicitement l'indice 0 ;
- en donnant l'indice de fin, le premier ayant implicitement l'indice 1 ;

Sous VB, nous utiliserons des parenthèses pour encadrer les indices.

Voici la syntaxe générale de déclaration de tableau appelé *NomTableau* et contenant des éléments de type *type*, les indices allant de *IndiceDébut* à *IndiceFin* :

```
dim NomTableau(IndiceDébut to IndiceFin) as type
```

Voici par exemple deux tableaux de 50 entiers :

```
dim Tab1(1 to 50)   as integer        '50 entiers de Tab1(1) à Tab1(50)
dim Tab2(73 to 122) as integer        '50 entiers de Tab1(73) à Tab1(122)
```

Si vous ne spécifiez pas l'indice de début, il sera par défaut de 0.

Ainsi, la déclaration

```
dim Tab3(50) as integer
```

revient à :

```
dim Tab3(0 to 50) as integer
```

Le tableau contiendra donc 51 éléments, accessibles de Tab3(0) à Tab3(50).

Vous pouvez paramétrer VB pour que le premier élément du tableau ait l'indice 1 par défaut. Il faut utiliser l'instruction *Option base 1* qui se met dans la partie déclaration d'un module comme la célèbre *Option Explicit*.

Ainsi :

```
Option base 1
dim Tab4(50) as integer
```

revient à :

```
dim Tab4(1 to 50) as integer
```

Conseil :

Bon, je vous conseille de toujours utiliser la syntaxe explicite en mentionnant les deux bornes. C'est plus clair.

4 E 2. Tableaux à plusieurs dimensions

Si l'on veut un tableau à plusieurs dimensions, on sépare ces dernières par des virgules :

```
dim NomTableau(IndDébut1 to IndFin1, IndDébut2 to IndFin2,…, IndDébutn to IndFinn) as type
```

Voici un tableau à deux dimensions :

```
dim t2(1 to 10, 1 to 10)  as integer
dim t3(-5 to 5, 50 to 60) as boolean
```

La borne implicite est toujours possible. Par exemple, *t2* peut être déclaré ainsi :

```
option base 1
dim t2(10, 10) as integer
```

L'accès à un élément du tableau est identique à la notation algorithmique : on met le nom du tableau puis l'indice voulu entre parenthèses.

Par exemple, *t(20)* correspond à l'élément d'indice 20 du tableau *t*.

Exercice 13

Continuons notre programme console avec les clients. Définissez un tableau contenant 10 clients.

4 F. Portée des variables, types, procédures, fonctions…

Tout élément déclaré (variable, type, procédure ou fonction) peut être local ou global. Qu'est-ce que cela signifie ? Tout dépend de l'endroit de la déclaration.

4 F 1. Variable déclarée dans un sous-programme

Toute variable déclarée dans un sous-programme sera forcément locale à ce sous-programme : son existence commence au début de l'exécution du sous-programme et s'achève lorsque le sous-programme se termine[1]. On ne peut donc s'en servir que dans le sous-programme.

Une telle variable est technique, à savoir ne sert qu'à réaliser le traitement du sous-programme. Un exemple classique ? Eh bien, un indice de boucle, une variable temporaire.

Ces variables se déclarent directement avec le mot clé *dim*, aucune option n'est possible.

4 F 2. Déclarations dans le module

Tout le reste se déclare dans le module : les variables, les types et les différents sous-programmes. En prenant le concept algorithmique classique, tous ces éléments seront globaux, sous-entendu au module, ce qui signifie que toute variable du module est accessible dans tout le module et tout sous-programme du module peut être appelé dans le module.

En algorithmique, un élément global est donc accessible dans l'ensemble du code qui est dans un seul fichier. Le problème, c'est que la programmation VB utilise plusieurs fichiers (modules). On distinguera donc :

- les éléments locaux du module (on parlera d'éléments privés) qui ne sont accessibles que dans le module qui les déclare ;
- les éléments globaux du module (on parlera d'éléments publics) qui sont accessibles dans tous les modules de l'application.

Par défaut, tout élément est public. Pour déclarer un élément en privé, il faut le faire explicitement.

[1] Je ne résiste pas au plaisir de vous citer un passage du livre *Poésie de l'informatique* de Milan Goldoniok :

Oh, variable locale à la procédure,
Tellement éphémère est ton règne,
Sitôt commencée, sitôt terminée la procédure,
Sitôt née, sitôt morte la variable.

4 F 3. *Public* et *private*

Nous avons établi que les variables locales à un sous-programme ne peuvent être, justement, que locales à ce sous-programme. On ne précise donc jamais leur portée.

Pour définir un élément local au module (donc utilisable dans le module où est la déclaration uniquement), on précède sa déclaration du mot clé *private*.

Pour définir un élément global au projet (donc utilisable dans tous les modules du projet), on précède sa déclaration du mot clé *public*.

Lorsque l'on précise la portée d'une variable avec l'un de ces mots clé, il remplace *dim*.

Si la portée de l'élément n'est pas définie, il sera public donc accessible dans tout le projet.

Les règles de programmation conseillent de ne rendre les éléments publics que lorsque c'est nécessaire. Il est donc préférable de mettre systématiquement *private*, sauf de façon motivée lorsque l'on veut que l'élément soit global.

Reprenons notre programme avec le client. Comme je n'ai qu'un module, des déclarations publiques n'ont pas trop de sens. Je déclare donc tout en privé :

```
01 Option Explicit
02
03 Private Type Client
04     Num As Integer
05     Nom As String
06     Prénom As String
07     Adr As String
08     Tél As String
09     DateClient As Date
10  End Type
11
12  Private i As Integer, j As Integer, Cl As Client
13  Private TabCl(1 To 10) As Client
14
15 Private Sub main()
16    With Cl
17       .Num = 0
18       .Nom = ""
19       .Prénom = ""
20       .Adr = ""
21       .Tél = ""
22       .DateClient = 0
23    End With
24 End Sub
```

Quelques explications suivent :

- ligne 3, le type *Client* est privé ;
- les différentes variables sont privées. J'ai donc remplacé lignes 12 et 13 l'instruction *Dim* par *Private* ;
- ligne 15, le programme principal (procédure *Main*) est maintenant privé.

4 G. Les opérateurs

Il n'y a rien à dire. On retrouve les opérateurs algorithmiques :

- opérateurs arithmétiques pour les calculs : +, -, *, /, mod, \[1] ;
- opérateurs logiques : *et*, *ou* et *non* deviennent *and*, *or* et *not*.

Les comparateurs (<, >, <=, >=, <>, =) sont toujours là.

[1] \ et *mod* correspondent à la division entière. \ est le quotient, *mod* le reste. Par exemple :
- 5 \ 2 = 2 ;
- 5 mod 2 = 1.

5. Les instructions de base

5 A. Affichage

L'instruction algorithmique :

```
Afficher expression1, expression2…, expressionn
```

se traduit par

```
MsgBox (expression1 & expression2 & expressionn)
```

En fait, *MsgBox* ne prend qu'un paramètre à afficher. Pour en afficher plusieurs, je les concatène avec l'opérateur &. Cette instruction possède d'autres paramètres (voir l'aide).

Par exemple, les instructions suivantes sont équivalentes :

```
MsgBox ('Bonjour' & ' les ' & 'amis');
MsgBox ('Bonjour les amis')
```

Ne vous inquiétez pas si ce n'est pas clair, dès que vous ferez un programme, et cela ne va pas traîner, cela deviendra évident.

Notez dès maintenant que tous les appels de sous-programmes internes à VB ont leurs paramètres mis entre parenthèses.

Nous verrons que les sous-programmes définis par l'utilisateur ne **doivent pas** avoir leurs paramètres encadrés de parenthèses quand on les appelle.

5 B. Saisie

L'instruction

```
saisir chaîne, variable
```

se traduit par :

```
variable = InputBox (chaîne)
```

Vous aurez noté que l'instruction algorithmique *Saisir* devient une fonction sous VB.

Exercice 14

Écrivez et testez un programme VB permettant de saisir puis d'afficher le nom de deux clients :

- un client stocké dans une variable indépendante *Cl* de type *Client* ;
- le 5e client de notre tableau *TabCl*.

Vous devez évidemment partir de notre programme console de l'exercice précédent. Les lignes de code doivent être écrites entre le *Private Sub Main* et le *End Sub*.

5 C. Affectation

On utilise le caractère « = », à ne pas confondre avec le comparateur « = » (ces deux mots clé sont donc homonymes).

6. Instructions de contrôle

En algorithmique, on a toujours un mot de début et un de fin pour chacune de ces instructions : *si… fin si*, *tant que… fin tant que*, *pour… fin pour*.

Ce principe est le même avec VB.

Nous allons voir tout cela.

6 A. Les tests

6 A 1. le *si*

```
If booléen then
        instruction1
        instruction2
       …
        instructionn
   else
        instruction1
        instruction2
       …
        instructionn
End If
```

Notez que :

- le mot clé *then* doit être sur la même ligne que le *if* ;
- la branche *else* est facultative.

✎Exercice 15✎

Sachant que les chaînes de caractères se comparent exactement comme des nombres (c'est l'ordre alphabétique qui est utilisé), modifiez le programme précédent pour qu'il vous demande deux noms (celui du client autonome et celui du 5e du tableau) puis qu'il affiche les deux noms dans l'ordre alphabétique.

6 A 2. Le *selon cas*

C'est l'instruction *select case*. Je vous laisse le soin d'étudier l'aide pour découvrir sa syntaxe.

6 B. Les boucles

Nous sommes encore très proches de l'algorithmique. Ce n'est donc pas très difficile.

6 B 1. Pour

L'instruction algorithmique

```
pour variable de entier1 à entier2 pas entier3
   instruction1
   instruction2
   …
   instructionn
fin pour
```

se traduit presque mot à mot en VB ainsi :

```
for variable = entier1 to entier2 step entier3
   instruction1
   instruction2
   …
   instructionn
next
```

Si le pas de la boucle est 1, vous pouvez omettre « step 1 ».

✎Exercice 16✎

Écrivez un programme affichant successivement les nombres 2, 4, 6, 8 puis 10.

6 B 2. Répéter

L'instruction algorithmique

```
répéter
   instruction1
   instruction2
   …
   instructionn
jusqu'à booléen
```

se traduit en VB comme suit :

```
Do
   instruction1
   instruction2
   …
   instructionn
Loop Until booléen
```

6 B 3. Tant que

L'instruction algorithmique

```
tant que booléen faire
   instruction1
   instruction2
   …
   instructionn
fin tant que
```

se traduit en VB ainsi :

```
Do While booléen
      instruction1
      instruction2
      …
      instructionn
Exit Do
```

6 B 4. Des boucles, j'en veux plus, j'en veux encore !

VB est l'ami des boucles. En effet, il propose toutes les combinaisons possibles : outre les traditionnelles *répéter jusqu'à* et *faire tant que*, vous avez accès aux non conventionnels mais parfois utiles *répéter tant que* et *faire jusqu'à*. Eh oui !

En fait, il suffit de retenir que les instructions *Loop* et *Do* peuvent être suivies de *While* ou *Until*.

Voici les syntaxes correspondantes.

L'instruction algorithmique qui n'existe pas (mais pourrait exister, d'ailleurs pourquoi n'existe-t-elle pas sinon à cause du manque d'imagination des informaticiens ?)

```
répéter
   instruction1
   instruction2
   …
   instructionn
tant que booléen
```

se traduit en VB comme suit :

```
Do
   instruction1
   instruction2
   …
   instructionn
Loop While booléen
```

L'instruction algorithmique qui n'existe pas (idem)

```
jusqu'à booléen faire
   instruction1
   instruction2
   …
   instructionn
fin jusqu'à
```

se traduit en VB ainsi :

```
Do Until booléen
      instruction1
      instruction2
      …
      instructionn
Exit Do
```

7. Les sous-programmes

Les procédures et les fonctions s'écrivent avant le programme principal (*sub Main*) partant du principe qu'il est plus logique de déclarer avant d'utiliser. C'est donc un conseil et non une obligation.

7 A. Paramètres

Chaque paramètre est suivi de « as » et de son type puis d'une virgule (« , ») si l'on a d'autres paramètres. On met le mot-clé *ByVal* si le paramètre est passé par valeur. (Par défaut, il est passé par adresse. Le mot clé *ByRef* explicitant le passage par adresse est donc facultatif.)

Exemple :

(a as integer, ByVal i as integer, ByVal j as integer, k as integer, l as integer)

Ici, *a*, *k* et *l* sont passés par adresse (j'omets le *ByRef* qui n'apporte rien) et *i* et *j* par valeur.

J'insiste sur le fait qu'il n'est pas possible de regrouper le mode de passage ou le type du paramètre. Ainsi,

(ByVal i, j as integer, ByRef a, k, l as integer)

n'a rien à voir avec la déclaration précédente puisque ici :

- *i* est un variant passé par valeur ;
- *j* est un entier passé par adresse ;
- *a* et *k* sont des variants passés par adresse ;
- *l* est un entier passé par adresse.

Soyez donc très prudent sur ces déclarations : elles ne correspondent pas à ce que vous pensez ou souhaitez mais elles ne produiront pas d'erreur puisqu'elles sont valides.

Lorsque vous passez un tableau en paramètre, vous ne devez pas mentionner sa taille. Vous écrirez donc par exemple *ByRef T() as integer*. Les tableaux ne peuvent pas être passés par valeur.

7 B. Procédures

On remplace *procédure* par *sub* et c'est tout ! Vous définirez les paramètres comme ci-dessus et n'oublierez pas de précéder la déclaration de *private* ou *public*.

La syntaxe sera :

```
Private Sub Nom (paramètres)
   Dim
      // variables locales

   instruction1
   instruction2
   …
   instructionn
End Sub
```

La procédure commence par *Sub* et se termine par *End Sub*. Vous noterez que j'ai sauté une ligne entre les déclarations de variables et le code. Ce n'est pas obligatoire mais améliore la lisibilité.

✎Exercice 17✎

Dans notre petit programme console, écrivez une procédure permettant de saisir un client quelconque (l'utilisateur doit saisir une valeur pour chaque champ).

✎Exercice 18✎

Écrivez une autre procédure permettant d'afficher un client passé en paramètre. Écrivez ensuite un programme principal testant la procédure de saisie et celle d'affichage.

7 C. Fonctions

7 C 1. La syntaxe

Pour la traduction anglaise, ce sera *function* au lieu de *fonction* ; la variable résultat sera le nom de la fonction et non plus *Résultat*. Pour les paramètres, voir 7.A !

La syntaxe sera :

```
Private Function Nom (paramètres) as type
   dim
      // variables locales

   instruction1
   instruction2
   …
   instructionn
   Nom = …
End Function
```

Attention, initialiser le nom de la fonction pour déterminer son résultat est une convention. Le nom ne correspond pas à une vraie variable. Je vous conseille donc de ne faire qu'une affectation au nom de la fonction, quitte à utiliser une variable intermédiaire pour calculer le résultat à renvoyer.

7 D. Appel d'un sous-programme

7 D 1. La fonction

Pour appeler une fonction, on utilise son nom et on liste ses paramètres entre parenthèses.

Voici un exemple (la fonction *Teckel* ne fait pas grand-chose mais là n'est pas la question) :

```
Private Function Teckel(i As Integer, j As Boolean) As Integer
    If j Then
          Teckel = i
       Else
          Teckel = 2 * i
    End If
End Function

Private Sub main()
   Dim i As Integer

   i = Teckel(3, True)
   MsgBox (Teckel(i, False))
End Sub
```

7 D 2. La procédure

VB propose deux techniques pour appeler une procédure définie par l'utilisateur :

- avec le mot clé *call*, le nom de la procédure et les paramètres sans parenthèse ;
- avec le nom de la procédure et les paramètres entre parenthèses.

Démonstration.

```
Private Sub Apprécier(Race As String, b As Boolean)
   If b Then
         MsgBox ("J'aime les " & Race)
      Else
```

```
         MsgBox ("Les " & Race & " ? Bof...")
   End If
End Sub

Private Sub main()
   Call Apprécier("Teckels", True)
   Apprécier "Cafards", False
End Sub
```

✎Exercice 19✎

Écrivez une fonction renvoyant le nombre de jours écoulés depuis qu'un client passé en paramètre est client. Vous aurez besoin de la fonction *Date* renvoyant la date du jour. Vous devrez sans doute aller consulter l'aide pour savoir comment réaliser des calculs sur les dates. Testez votre fonction dans notre programme précédent.

7 E. Les limitations

Certaines fonctions algorithmiques deviennent des procédures sous VB. En effet, une fonction VB ne sait renvoyer qu'un type de base (entier, réel, booléen…). Pour renvoyer des tableaux ou des structures, on utilise raisonnablement une fonction en algorithmique mais il faut écrire une procédure en VB, cette dernière possédant au moins un paramètre passé par adresse pour représenter le résultat.

✎Exercice 20✎

Ah, nous avons oublié de travailler sur notre tableau de clients. Je vous demande donc d'écrire sous VB (toujours dans le même programme) :

- une procédure *SaisieClientTableau* permettant de saisir un client du tableau (lequel ? ce sera un paramètre) ;
- une procédure *AffichageClientTableau* affichant un client du tableau (lequel ? ce sera un paramètre).

Vous testerez vos sous-programmes en saisissant puis en affichant le 5^{e} client du tableau.

8. Appliquons tout cela

8 A. Le cadre de travail est modeste

Je vous propose d'écrire un petit programme utilisant un module que nous allons également créer (le projet contiendra donc deux modules). Il ne s'agit ici que d'une mise en pratique de la syntaxe VB que nous venons d'apprendre. Le module sera donc assez artificiel, mais cette application vous permettra d'en écrire de plus vraisemblables dans les séquences à venir.

Nous allons d'abord écrire un programme complet pour ensuite en transformer une partie en module utilisé par le programme. L'important, je le répète, c'est la démarche plus que la réalisation elle-même.

Vous pouvez donc sauvegarder votre programme gérant les clients, nous ne nous en servirons plus dans cette séquence.

8 B. Le programme

Lorsque vous affichez une chaîne de caractères avec l'instruction *MsgBox*, elle s'affiche assez sobrement :

```
writeln ('Salut')
```

s'affichera ainsi :

Je suis en train d'écrire un programme affichant pas mal de choses et j'aimerais disposer d'un affichage plus visuel. Nous serions en programmation événementielle, la question ne se poserait pas puisque je pourrais changer la couleur, la police, la taille ou la mise en forme de mon texte sans problème. Dans cette séquence, nous nous limitons à une application console ; je n'ai donc aucun effet visuel à ma disposition. Je vais faire le maximum en ajoutant une étoile avant et après mon texte (un espace séparant les étoiles et le texte pour plus de lisibilité).

Je vais donc écrire une procédure *MsgBoxÉtoile* qui, lorsqu'on écrira :

```
MsgBoxÉtoile ('Salut')
```

affichera à l'écran

Notez qu'un paramètre devant être d'un type donné, ma procédure ne pourra afficher qu'un seul type de données. Je choisis d'afficher des chaînes de caractères.

Encore une fois, je vous concède que cette mise en relief est d'un autre âge. S'il y a bien un domaine dans la vie où l'on ne peut pas dire sans monstrueuse hypocrisie que c'était mieux avant, c'est bien l'informatique[1].

✎Exercice 21✎

Finalement, ce n'est pas moi, mais vous qui allez écrire cette procédure (c'est trivial). Écrivez ensuite un programme appelant cette procédure et affichant un texte saisi par l'utilisateur en l'encadrant d'étoiles.

Finalement, mon affichage est sympathique, mais je sens ma fibre artistique réclamer encore plus. Mon objectif est de transformer *MsgBoxÉtoile* pour obtenir ce genre de résultat :

```
MsgBoxÉtoile ('Salut')
```

affichera à l'écran

✎Exercice 22✎

Alors là, c'est un peu moins simple à réaliser car le nombre d'étoiles à afficher en haut et en bas dépend de la longueur de la chaîne passée en paramètre. Je vous donne donc une information et une consigne :

- VB possède une fonction *len (ch : string) : integer* prenant en paramètre une chaîne de caractères et renvoyant sa longueur ;
- je vous demande d'écrire une fonction *LigneÉtoile* renvoyant une chaîne constituée d'un nombre d'étoiles passé en paramètre. Vous utiliserez *LigneÉtoile* dans *MsgBoxÉtoile* en concaténant les lignes d'étoiles au message à afficher.

Vous testerez tout cela avec votre programme précédent.

Bien.

L'affichage obtenu est tellement sublime que j'ai envie de m'en servir dans tous mes programmes console. Cela sera en quelque sorte ma *griffe*. Comment faire ? Il y a deux techniques :

- par un copier/coller, j'ajoute systématiquement les deux procédures *LigneÉtoile* et *MsgBoxÉtoile* à tous mes programmes. C'est un peu fastidieux et, si je veux un jour modifier ce style d'affichage pour réaliser un encadrement double par exemple, je devrai reprendre les programmes un à un ;
- seconde solution, je vais créer un module *Étoile* contenant mes deux procédures. Tout projet voulant s'en servir n'aura qu'à référencer ce module. Pour modifier mon encadrement, je n'aurai qu'à modifier le nouveau module (et tout de même à recompiler les programmes qui s'en servent). Attention aux priorités !

Je ne vous cache pas que nous allons mettre en œuvre la seconde solution.

[1] Quoique, en se forçant, on y arrive : l'informatique sous Dos, très rustique, plantait beaucoup moins que les savantes horlogeries, subtiles et souvent déréglées, de la programmation Windows.

8 C. Le module

✎Exercice 23✎

Ajoutez un nouveau module au projet et déplacez-y le code nécessaire. Le programme s'en trouve nettement raccourci !

8 D. Mise à jour de l'unité

Je souhaite maintenant que l'instruction `MsgBoxÉtoile ('Salut')` affiche à l'écran :

✎Exercice 24✎

Modifiez le module *Etoile* en conséquence et recompilez le programme : cela marche.

Voilà. Vous savez créer, utiliser et modifier un module. Nous nous en resservirons plus tard.

9. Compléments

9 A. Débuggeur

On retrouve les fonctionnalités habituelles : la pose de point d'arrêt, l'évaluation ou la modification de variables en cours d'exécution, les modes pas à pas et pas à pas approfondi. Tout cela se trouve dans le menu *Débogage*.

Des outils encore plus pointus sont également disponibles dans ce menu : points d'arrêt conditionnels, points de suivi… nous verrons cela dans la séquence suivante.

9 B. Exercice d'application

Exercice 25

Vous allez écrire un programme résumant toute cette séquence. Il s'agit de trier un tableau de vingt entiers. Je vous demande de faire des sous-programmes pour :

- saisir le tableau. Pour pouvoir tester rapidement le programme, vous initialiserez les éléments avec des valeurs aléatoires en utilisant la fonction *Rnd* (voir l'aide VB pour la syntaxe) ;
- afficher le tableau ;
- trier le tableau. Je vous laisse le choix de la méthode. Comme vous êtes sensé connaître tout ce qui est algorithmique, ce savoir ne doit pas vous échapper.

N'hésitez pas à reprendre les sous-programmes des exercices précédents si vous en avez besoin. N'oubliez pas qu'il faut impérativement définir un type tableau.
Faites tout dans un seul module.

Séquence 6

Le débuggeur

Le débuggeur est un outil indispensable que nous allons découvrir maintenant.

Contenu

Le débuggeur

Capacités attendues

Savoir l'utiliser

1. Le débuggeur

1 A. Les phases finales du cycle de développement

Le préfixe *dé* est privatif. Ainsi, *dé*faire signifie « Changer une chose de manière qu'elle cesse d'être faite (*Le petit Robert*) ».

De la même façon, *dé*buguer signifie enlever les bugs. L'outil adapté à cette fonction est le débuggeur[1].

Au fait, qu'est-ce qu'un bug ? Je vous passe la justification historique du terme. En informatique moderne, un bug est un terme poli pour évoquer une erreur de conception. Il y en a de toutes sortes :

- une instruction oubliée (on ne contrôle pas si le tableau est vide avant de le manipuler) ;
- un test erroné (*i <= 0* au lieu de *i < 0*) ;
- l'accès possible à une commande depuis l'interface graphique alors que cette commande est illogique dans le contexte ;
- …

Pour résumer, tout ce qui entraîne un disfonctionnement du programme par rapport au cahier des charges est un bug.

Si l'on emploie un terme poli (*bug* est moins agressif qu'*erreur*), c'est parce que le bug est assez fréquent. Même un très bon programmeur en fait. Allons plus loin encore : lorsque les erreurs les plus évidentes ont été corrigées, il devient trop coûteux de chercher les bugs pointus. On livre alors le produit au client qui, à l'usage, détectera les bugs. Vous développerez ensuite des *patchs* (mises à jour) corrigeant les bugs détectés (et hélas, en introduisant parfois d'autres).

C'est la même chose pour ce cours que vous êtes en train de lire. Après l'avoir écrit, je l'ai relu attentivement trois fois. Au vu du nombre de pages, vous reconnaîtrez que ces trois relectures sont un travail en soi ! À quoi servent-elles ? À débusquer toutes les erreurs :

- algorithmes faux (contenant des bugs) ;
- fautes de français (orthographe, grammaire) et phrases peu claires ;
- renvois erronés…

À la première relecture, je fais quatre ou cinq corrections par page. À la seconde, une par page. À la troisième, une toutes les deux à trois pages.

Si je faisais une quatrième relecture, je trouverais certainement une faute toutes les dix pages (d'autant qu'il arrive fréquemment que la correction d'une erreur en entraîne une autre). Je ne fais pas cette nouvelle relecture car j'estime qu'elle n'est pas rentable : le temps que j'y passerais est trop long pour ne trouver qu'une quarantaine de fautes sur l'ensemble du cours.

Attention, le fait qu'il y ait ne serait-ce qu'une faute dans tout le support m'ennuie. Mais bon. Avoir quatre ou cinq fautes par page, c'est lamentable. En avoir une toutes les dix pages, c'est une coquille tout à fait excusable.

Revenons à notre programme. Sous traitement de texte, on parlera de coquilles (et non d'erreurs) si elles sont raisonnablement peu nombreuses. De même, dans un programme, on parlera de bugs et non d'erreurs si, dans l'ensemble, le programme fonctionne correctement. En revanche, si un traitement central de l'application est manifestement erroné, cela ne va pas du tout.

[1] En français, on parle de *bogues* que l'on élimine en *déboguant* le programme avec un *débogueur*. C'est le *débogage*.

Qu'importe si votre première version de l'application est bourrée d'erreurs ! L'important, c'est que la version finale soit au maximum dépourvue de bugs.

Comme la chasse au bugs est assez rébarbative, mieux vaut programmer de façon rigoureuse pour introduire le moins possible d'erreurs :

- tester sur papier les traitements complexes ;
- définir très soigneusement ses tests (alternatives et boucles) ;
- …

Lors de l'exécution, vous avez deux techniques pour corriger un bug qui se révélerait :

- vous prenez votre code source et vous vous cassez la tête dessus ;
- vous réalisez une exécution encadrée (contrôlée) du programme pour observer de l'intérieur comment il fonctionne.

La seconde solution, évidemment la plus simple, met en œuvre le débuggeur. Nous allons étudier ce sympathique outil. Attention, comprenez bien que cela ne concerne que les erreurs d'exécution et non celles de syntaxe empêchant la compilation du programme.

1 B. Les fonctionnalités du débuggeur

Je viens de dire que le débuggeur permettait une exécution encadrée d'un programme. Qu'est-ce que cela signifie au juste ?

Eh bien, l'exécution normale d'un programme consiste à le lancer et c'est tout. Si l'on veut connaître la valeur d'une variable ou vérifier que l'on passe bien par un sous-programme ou le corps d'une alternative, on n'a pas d'autre choix que d'ajouter dans le source des instructions *MsgBox ("Coucou")* là où on veut vérifier notre passage et *MsgBox ("i = " & i)* pour connaître la valeur de *i* à un moment précis. Il faut alors relancer le programme pour vérifier l'affichage des *Coucou* et du contenu des variables.

Avec l'exécution encadrée proposée par le débuggeur, on accède au source du programme en même temps que celui-ci est exécuté. On peut alors en toute liberté connaître en direct la valeur de chaque variable, modifier sa valeur et suivre l'exécution ligne à ligne du programme.

Voici ce que le débuggeur propose :

- la pose de points d'arrêt. Si vous posez un point d'arrêt sur une instruction et lancez le programme, l'exécution s'interrompra lorsque l'application atteindra cette instruction. Vous pouvez alors observer voire modifier la valeur des variables, passer en mode pas à pas… ;
- le mode pas à pas revient à poser un point d'arrêt sur chaque instruction. Vous avez la main après l'exécution de chacune d'elles. Cela permet notamment de voir l'enchaînement des instructions exécutées ;
- les espions, permettant de suivre la valeur des différentes variables ou de les modifier.

2. Utilisation du débuggeur

Le débuggeur est un outil systématiquement associé aux compilateurs modernes (C, Delphi, Java, VB…). Je vais vous présenter celui de VB 6, sachant que vous retrouverez exactement les mêmes fonctionnalités dans tous les autres langages.

Le débuggeur est toujours disponible, aucune manipulation particulière n'est nécessaire.

2 A. Le programme support

Créez un nouveau projet et faites-en une application console. Tapez alors le code suivant dans le module.

Voici le programme :

```
Option Explicit

Sub Init(T() As Single)
   Dim i As Integer

   For i = 1 To 50
      T(i) = Rnd
   Next
End Sub

Sub Compte(Inf05 As Integer, Sup05 As Integer, T() As Single)
   Dim i As Integer

   Inf05 = 0
   Sup05 = 0
   For i = 1 To 50
      If T(i) < 0.5 Then
            Inf05 = Inf05 + 1
         Else
            Sup05 = Sup05 + 1
      End If
   Next
End Sub

Sub Main()
   Dim Nbr1 As Integer, Nbr2 As Integer
   Dim T(1 To 50) As Single

   Call Init(T)
   Call Compte(Nbr1, Nbr2, T)
   MsgBox ("Nbr < 0,5 = " & Nbr1 & "   Nbr > 0,5 = " & Nbr2)
End Sub
```

Notez que c'est une application console donc le programme principal est la procédure *Main* (en gras).

Que fait ce programme ? Rien de bien original, il remplit un tableau de 50 valeurs réelles aléatoires comprises entre 0 et 1 (procédure *Init*) puis affiche les nombres de valeurs inférieures à 0,5 et supérieures à 0,5 du tableau (procédure *Compte*).

Arrivé à ce stade de votre formation, vous devez comprendre ce source sans aucune difficulté.

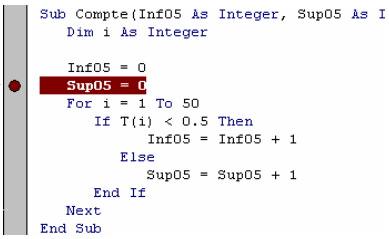
Exécutez le programme pour vérifier qu'il fonctionne. La somme des deux nombres affichés doit être égale à 50.

2 B. Pose de points d'arrêt

Pour poser un point d'arrêt, c'est tout simple : vous positionnez le curseur de l'éditeur sur l'instruction concernée puis vous exécutez *Débogage/Basculer le point d'arrêt* (ou alors raccourci clavier *F9*).

Posez un point d'arrêt sur une instruction quelconque. Il sera explicitement signalé par une ligne et une pastille rouge :

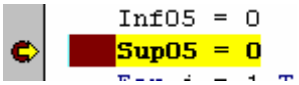
Pastille matérialisant le point d'arrêt



Une troisième façon de poser un point d'arrêt consiste à cliquer dans la bande verticale grise à gauche du source (là où est actuellement la pastille) sur la ligne d'une instruction.

Pourquoi la commande s'appelait *Basculer* et non simplement *Pose* ? Tout simplement car pour enlever un point d'arrêt, il suffit de répéter la manœuvre de pose. Vous êtes sur une instruction, faites *F9*, le point d'arrêt est posé, refaites *F9*, le point d'arrêt est supprimé.

Exécutez le programme avec le même point d'arrêt que moi. Que se passe-t-il ? On ne le voit sans doute pas très bien sur cette copie d'écran monochrome :



mais sur votre écran, cela doit être visible. En surimpression du point d'arrêt en rouge, il y a du jaune fluo. La pastille contient une flèche jaune.

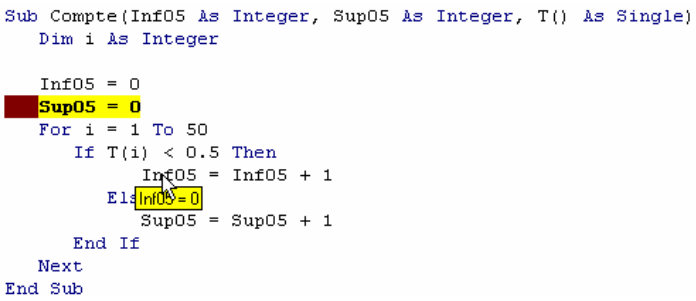
Le surlignage en jaune fluo avec sa petite flèche indique la ligne (l'instruction) qui va être exécutée. Ici, on vient d'exécuter *Inf05 = 0* et on va exécuter *Sup05 = 0*. Le point d'arrêt a joué son rôle : il a interrompu l'exécution.

2 C. Évaluation et modification des variables

Comme l'exécution est suspendue, j'ai accès au programme source et à la mémoire. Cela me permet de connaître ou de modifier la valeur des variables.

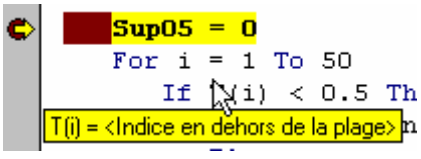
2 C 1. Info-bulle

Pour avoir rapidement la valeur d'une variable, il me suffit de placer le curseur dessus et d'attendre quelques instants. Sa valeur s'affiche dans une info-bulle :



Je n'ai évidemment accès qu'aux variables disponibles là où je suis (soit ici aux variables locales à la procédure et à ses paramètres).

Mettez votre curseur sur le *T* de *T(i)* pour obtenir sa valeur. Voici ce que vous obtenez :



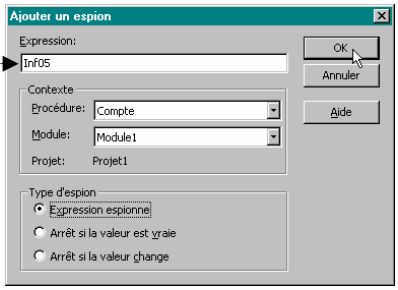
Que se passe-t-il ? En fait, le compilateur a compris que vous vouliez la valeur de l'expression *T(i)*. Il vous affiche un message d'erreur. Pourquoi ? Car *i* valant actuellement 0 (vérifiez-le), *T(i)* c'est *T(0)*. Comme j'ai déclaré l'indice de *T* allant de 1 à 50, la valeur 0 n'est pas valide. Le message d'erreur vous informe que *i* est hors de la plage autorisée (qui va de 1 à 50).

2 C 2. L'espion

L'info-bulle est pratique pour avoir ponctuellement et rapidement une valeur. On peut mettre en place un mécanisme plus lourd, l'espion. Il va espionner (suivre) une variable tout au long du programme en vous affichant en permanence sa valeur.

Pour placer un espion, cliquez sur la variable à espionner puis menu *Débogage/Ajouter un espion…* Vous obtenez cette fenêtre :

Si vous lancez la commande sans être sur une variable, vous n'avez qu'à remplir cette zone de texte.
C'est également utile si l'expression proposée n'est pas ce que vous souhaitez.



Je vous laisse le soin d'étudier les différents types d'espions. Celui qui m'intéresse est *Expression espionne*. Faites *OK*. Vous obtenez alors une nouvelle fenêtre *Espions* :

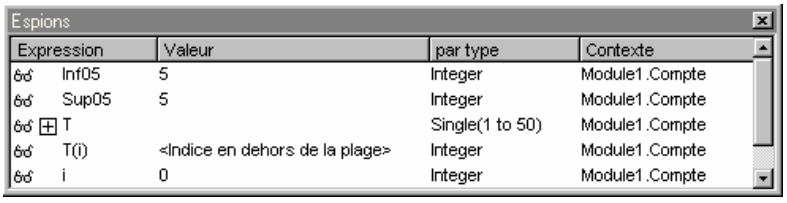
Cette fenêtre contient tous les espions du programme (pour le moment, il n'y en a qu'un). Pour changer la valeur de la variable espionnée, cliquez sur sa valeur (ici 0) et changez-la.

Faites les opérations suivantes :

- définissez un espion sur *Sup05* ;
- modifiez les valeurs d'*Inf05* et *Sup05* pour qu'elles valent 5 ;
- définissez un espion sur *i* ;
- définissez un espion sur *T(i)* ;
- définissez un espion sur *T*.

Vous aurez sans doute besoin de définir vous-même la valeur *Expression* dans la fenêtre *Ajouter un espion* pour certains d'entre eux. Autre technique : vous sélectionnez l'expression à espionner et vous définissez un espion express (raccourci *Shift+F9*).

Voici alors l'état de la fenêtre *Espions* :



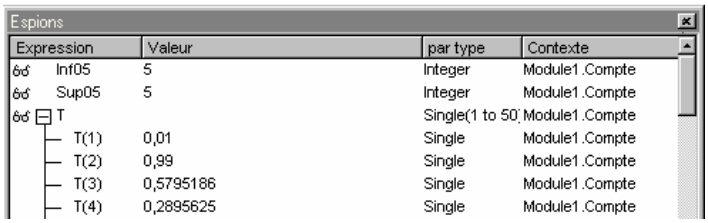
On retrouve que *T(i)* ne signifie rien lorsque *i* vaut 0.

Vous remarquerez que *T* ne possède aucune valeur mais est précédé par un petit +. Ce symbole, vous le connaissez déjà avec la même sémantique dans la partie gauche de l'explorateur présentant l'arborescence. Il indique ici que la variable *T* contient d'autres

variables (et c'est vrai puisque le tableau *T* contient 50 éléments). Pour avoir le contenu complet, vous pouvez cliquer sur le +.

Vous en profiterez pour changer les valeurs de *T(1)* et *T(2)* qui vaudront respectivement 0,01 et 0,99. Attention, il faudra taper ces valeurs au format américain, soit 0.01 et 0.99. La différence ? Un point et non une virgule pour indiquer les décimales.

Voici le résultat :



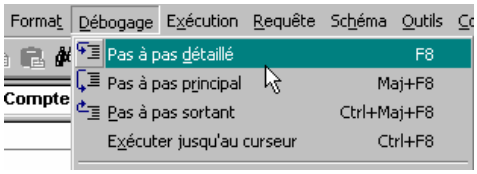
Vous pouvez réduire *T* en cliquant sur le − qui le précède. Vous retrouverez le + avec les variables structurées.

2 D. Avancer l'exécution de l'application

2 D 1. Mode pas à pas

Nous allons exécuter instruction par instruction la suite du programme. Observez que les espions sont fidèlement mis à jour.

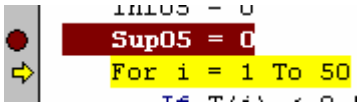
Les commandes sont dans le menu débogage :



Ces quatre modes d'avance sont assez parlants. Consultez l'aide pour plus de détails. Ce qui m'intéresse ici est le plus classique, le mode *Pas à pas détaillé* qui s'arrête sur chaque instruction. Notez son raccourci clavier *F8*.

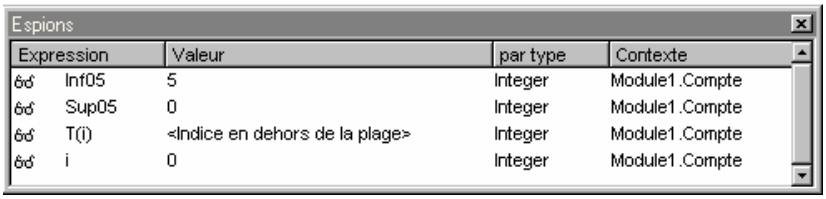
Appuyez donc sur *F8* et observez les changements.

Dans le source :



La ligne surlignée est descendue d'une instruction. Cela signifie que *Sup05 = 0* a été exécutée et que l'instruction en attente est maintenant la boucle *For*.

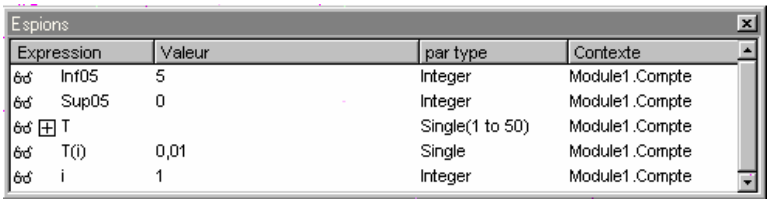
Dans la fenêtre *Espions* :



Les espions reflètent fidèlement la valeur de leur expression sous-jacente. On peut donc suivre au jour le jour (ou plutôt instruction après instruction) leur valeur.

On voit que *Sup05* vaut maintenant 0. C'est normal : il valait 5, puis on a exécuté *Sup05 = 0*. Sa valeur a donc été mise à 0.

Faites encore *F8*. L'exécution de la boucle *For* a lieu, ce qui initialise *i* à 1. L'espion *T(i)* peut alors afficher une valeur :



Continuez à exécuter l'application instruction par instruction, tout en surveillant d'un œil le code source et la ligne courante surlignée qui bouge (observez notamment ce qui se passe lorsqu'une itération de *For* a été faite) et d'un autre œil la fenêtre des espions. Vérifiez la mise à jour de *Inf05* et *Sup05* en fonction de *T(i)*.

2 D 2. Aller un peu plus vite

Vous avez sans doute réalisé quelques itérations de la boucle. Je ne pense pas que vous soyez allé jusqu'aux 50 prévues.

Lorsque les itérations vous ont permis de tester les deux cas possibles (valeur inférieure ou supérieure à 0,5), la suite de la boucle n'apporte rien.

Que faire alors pour accélérer le débogage ? Il y a plusieurs solutions :

- vous pouvez arrêter purement et simplement l'exécution du programme (par exemple, vous avez identifié une erreur, il faut donc la corriger). Dans ce cas, commande *Exécuter/Fin* (à gauche de la commande, vous avez le dessin du bouton correspondant dans la barre d'outils) ;
- si vous voulez continuer normalement l'exécution du programme (qui ne s'arrêtera alors plus, sauf s'il rencontre un nouveau point d'arrêt), *Exécuter/Continuer* (ou *F5* ou le bouton adéquat de la barre d'outils) ;
- si vous voulez terminer l'exécution de la boucle pour retrouver le mode débogage juste après, soit vous posez un point d'arrêt sur l'instruction suivante (ici le *End Sub*), soit vous cliquez sur cette ligne *End Sub* et faites *Débogage/Exécuter jusqu'au curseur* soit, dans le cas présent, faites *Débogage/Pas à pas sortant* puisque la fin de la boucle coïncide avec celle de la procédure.

3. Conclusion

Vous savez maintenant utiliser le débuggeur. Lorsqu'un de vos programmes ne fonctionne pas comme il le devrait, essayez de localiser l'erreur à la main. C'est très formateur !

Si cette recherche ne donne rien, exploitez le débuggeur.

Il est évident qu'un bon développeur doit parfaitement maîtriser l'usage de cet outil. N'hésitez donc pas à le mettre en avant lors de l'épreuve pratique. C'est bien utile pour illustrer le fonctionnement d'un programme.

Travaux dirigés 1 : travail sur l'interface

L'objet des exercices qui suivent est de mettre en œuvre de façon systématique les concepts vus dans le cours pour que cela devienne un automatisme.

Plus que de simples exercices d'application, ils sont plutôt un complément de cours. Ne les bradez pas !

Enfin, faites les exercices dans l'ordre, la difficulté étant croissante.

Résumons notre savoir des séquences 1 à 6 :

- nous savons manipuler les bases de l'interface graphique en créant une feuille et en manipulant les deux contrôles les plus simples (la zone de texte et le bouton) ;
- nous savons utiliser les événements, bien que nous n'en connaissions pour le moment qu'un petit nombre ;
- nous savons *grosso modo* ce qu'est un module et un objet. En tout cas, nous en connaissons suffisamment pour écrire du code et utiliser des objets ;
- nous connaissons les instructions VB permettant de coder un algorithme.

Nous allons travailler de nouveau sur notre programme avec les clients. Cette fois, nous ferons moderne avec une interface graphique.

Votre travail consistera avant tout à générer le formulaire et à faire du copier/coller du code que nous avions écrit dans les différents événements.

Exercice 1 : Affichage

Faites une interface graphique permettant à l'utilisateur de saisir les différentes caractéristiques d'un client.

Il vous faudra autant de zones de texte que de champs dans la structure.

Exercice 2 : Initialisation du client

Je souhaite maintenant que vous repreniez le type *Client* et une variable associée. Il suffit de faire un *copier* de notre ancien programme console pour le placer au bon endroit dans le code. Je veux ensuite que vous ajoutiez un bouton *Enregistrer* qui, lorsque l'on clique dessus, prend les différentes valeurs saisies dans le formulaire pour initialiser notre variable client.

Évidemment, l'utilisateur est sensé avoir rempli les différentes zones de texte (enfin, les *TextBox*) avant de cliquer sur le bouton. Vous ne ferez aucune vérification en ce sens.

Exercice 3 : Mise à zéro du formulaire et affichage du client

Je souhaite maintenant que vous ajoutiez deux nouveaux boutons :

- *Vider* doit mettre à zéro l'ensemble de la feuille, c'est-à-dire vider le contenu de chaque zone de texte ;
- *Afficher* doit afficher dans le formulaire la valeur de notre variable *Cl* (c'est donc l'inverse d'*Enregistrer*).

Exercice 4 : Affichage du nombre de jours

Ajoutez au formulaire un contrôle qui doit contenir, lorsque vous affichez le client, le nombre de jours depuis lequel il est client de l'entreprise.

Vous aurez besoin de la fonction VB *Date* renvoyant la date système (date du jour configurée sur l'ordinateur).

Exercice 5 : Problème d'ergonomie

Ce formulaire possède un problème : lorsque je saisis un client, je peux taper une valeur quelconque dans le contrôle du nombre de jours. Or, d'après mon sujet, ce contrôle ne doit pas servir à taper une valeur, mais uniquement à afficher un résultat venant d'un calcul réalisé par l'application.

Pourriez-vous corriger cela ? Une petite aide : il faut modifier une propriété du contrôle. Laquelle ? J'ai dit *petite* aide, pas solution !

Exercice 6 : Problème d'ergonomie bis

Dire depuis combien de jours un client est client, ce n'est pas forcément très explicite. J'aimerais mieux un affichage indiquant le nombre d'années, de mois et de jours.

Faites cette modification. Vous écrirez un sous-programme prenant un nombre de jours et renvoyant le nombre d'années, de mois et de jours correspondants. Vous changerez également l'étiquette correspondante.

Inutile de faire du zèle vis-à-vis des différentes années bissextiles ou du fait que certains mois ont plus de jours que d'autres.

Pour réaliser la correction, j'ai eu besoin d'utiliser l'opérateur de concaténation de chaîne de caractères. Vous en aurez peut-être besoin également. Eh bien, sous VB, c'est « & ».

Séquence 7

Les contrôles

Nous allons étudier les principaux contrôles visuels proposés par VB.

Contenu

Rôle des différents contrôles.

Exploiter les contrôles indépendants.

Capacités attendues

Savoir créer tout type de feuille.

1. Les contrôles de saisie

(Nous revenons dans le cadre des applications événementielles utilisant une interface graphique.)

1 A. Rappels

Nous avons vu que tous les objets et en particulier les contrôles possèdent des propriétés accessibles par l'inspecteur d'objets. Certaines propriétés sont communes à tous les contrôles, d'autres sont spécifiques à un seul. Voici quelques exemples de ce que l'on peut paramétrer pour chaque contrôle :

- sa taille et sa position dans la feuille (propriétés *Top*, *Left*, *Height* et *Width*). On ne modifie jamais à la main ces propriétés : c'est en déplaçant ou en redimensionnant le contrôle à la souris qu'elles sont mises à jour ;
- la couleur du fond ou de la police (propriétés *BackColor* et *ForeColor*) ;
- le fait qu'il soit visible ou non (propriété *Visible*) ;
- le fait que l'on puisse modifier le contenu de sa zone de saisie (propriété *Locked*[1]).

Lorsque vous posez un contrôle sur une feuille, ses différentes propriétés sont initialisées à des valeurs par défaut. (Par exemple, tous ont par défaut leur propriété *Visible* à *True* et *Locked* à *False*.) Vous devez alors modifier les propriétés selon vos souhaits pour obtenir le contrôle adapté à vos besoins.

Il est temps de rentrer dans le détail des contrôles. Pour chacun d'eux, nous allons voir :

- dans quel cas l'utiliser. En effet, chaque contrôle a une forme ou une caractéristique propre qui le rend plus efficace dans la saisie d'un type de données ; par exemple, pour saisir une valeur booléenne (*oui* ou *non*), on peut demander à l'utilisateur de taper au clavier soit *oui*, soit *non*. C'est lamentable, mais techniquement, c'est correct. On peut également utiliser une case à cocher, qui est le composant fait pour cela ;
- comment il fonctionne ;
- comment le paramétrer (c'est-à-dire remplir ses propriétés).

Je vous préviens que je ne compte absolument pas être exhaustif. Il est possible de faire plusieurs pages par contrôle. Mais ce que je dirais alors, vous le trouverez dans l'aide. Je ne vais donc donner que les éléments essentiels.

De même, nous n'allons pas voir tous les contrôles de VB, mais seulement ceux de base.

Vous **devrez** compléter ce cours par la pratique et l'étude de l'aide. Un exemple ? Chaque contrôle (en fait, chaque objet : feuille, contrôle…) possède de très nombreuses propriétés. Nous verrons celles qui sont essentielles, mais pour toutes les autres, vous devrez consulter l'aide pour découvrir leur rôle. L'idée n'est pas d'apprendre par cœur toutes les propriétés, mais de savoir trouver dans l'aide celles dont on a besoin.

Attention ! Si vous ne le faites pas, vous ne percevrez pas tout le potentiel des contrôles. Des choses qui vous sembleront impossibles à faire pourront l'être par des propriétés que vous ne connaissez pas. Comprenez bien que les propriétés caractérisent les objets. De façon naturelle, on peut donc se douter que toutes les caractéristiques intéressantes (utiles) sont représentées par des propriétés.

Notez que les contrôles ne sont pas inventés par VB : tous proviennent du système d'exploitation Windows. Ainsi, quelle que soit l'application Windows utilisée (Word ou une application VB que vous avez développée), on retrouve une interface uniforme. Lorsque vous voyez une liste

[1] *Locked* signifie *verrouillé*.

modifiable, vous savez que vous devez choisir une valeur parmi une liste (par exemple, la liste modifiable pour saisir une police sous Word). Lorsque vous voyez une case à cocher, vous savez que vous pouvez ou non activer quelque chose (par exemple l'exposant dans la boîte de dialogue *Police* sous Word).

1 B. Localisation des contrôles

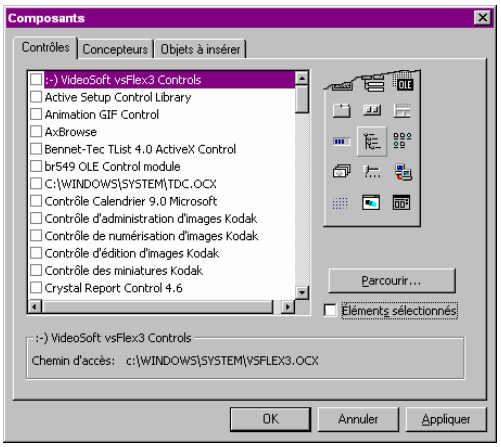
Les contrôles se trouvent dans la boîte à outils :

Cela en fait vingt.

Cela semble peu… en fait, c'est très peu ! Il en existe beaucoup d'autres puisque chaque développeur peut créer ses propres contrôles puis les exploiter sous VB. Une situation plus fréquente est que les applications proposent parfois des contrôles permettant d'accéder à leurs documents par programmation via un contrôle.

Ces contrôles supplémentaires sont des *ActiveX* (voir l'aide). Ils sont stockés dans des fichiers d'extension *ocx* normalisés. Ainsi, un contrôle ActiveX peut être utilisé par tout langage de programmation reconnaissant la norme : Delphi, VB…

Pour ajouter des contrôles à la boîte à outils, faites la commande *Menu/Composants…* Vous obtenez alors la boîte de dialogue suivante :

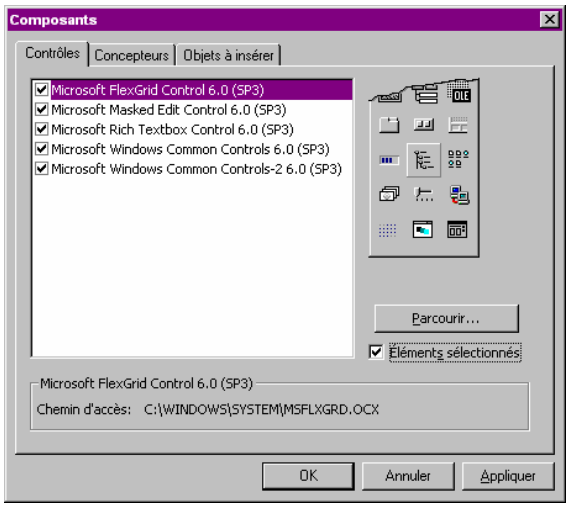


Vous n'avez qu'à cocher les cases qui vous intéressent (oui, les noms ne sont pas toujours très parlants) pour ajouter ceux qui vous intéressent. Avec *Parcourir*, vous pouvez ajouter n'importe

quel *ocx* présent sur votre disque. Notez que vous aurez peut-être sur votre écran une liste différente de la mienne : cela dépend un peu de la configuration de votre système.

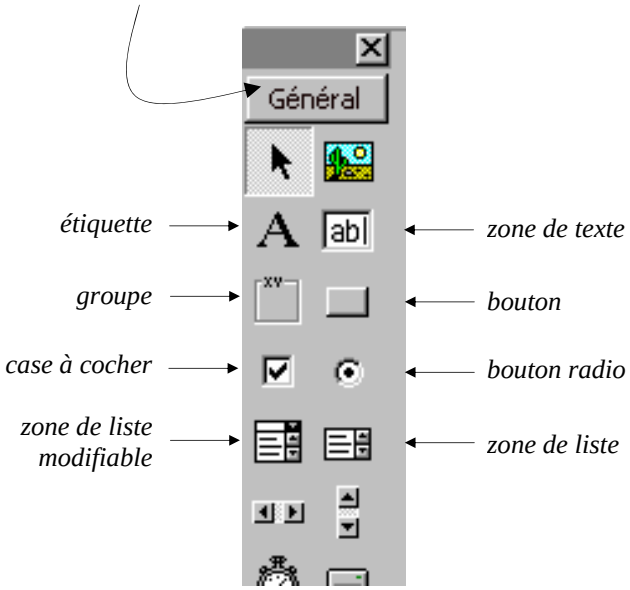
Allez faire un tour dans l'onglet *Objets à insérer* : cela vous permet d'ajouter des contrôles *Document Word*, *Classeur Excel*… L'intérêt ? Eh bien, au lieu d'avoir une bête zone de texte dans une feuille, vous pouvez disposer d'un document Word avec toutes les commandes Word de mise en page !

Voici une sélection de divers composants (un fichier *ocx* peut contenir plusieurs contrôles) de Microsoft que vous devez avoir avec VB 6. Je vous conseille de les ajouter et de les tester à vos moments perdus pour bien voir ce que vous avez à votre disposition. Nous n'allons étudier que les contrôles de base et ceux indiqués ci-dessous constituent souvent des variantes qui peuvent être extrêmement utiles.



1 C. Ceux que nous allons étudier

Testez le clic droit sur l'onglet Général*… Vous pouvez ajouter d'autres onglets pour structurer la boîte à outils si vous la dotez de nombreux contrôles.*



Les autres contrôles ne sont pas inintéressants… je vous laisse le plaisir de les étudier vous-même !

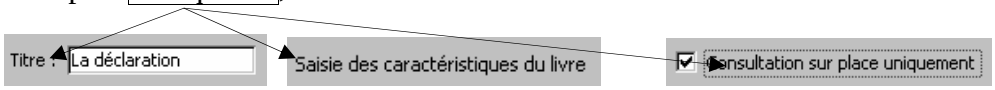
1 D. Étiquette (nom VB *Label*)

Si vous voulez écrire du texte dans une feuille, vous le ferez avec un contrôle *Étiquette* (*Label* sous VB).

La particularité de ce contrôle est qu'il ne permet aucune saisie (il n'est d'ailleurs associé à aucune zone de saisie). Il correspond aux champs pré-imprimés des formulaires papier.

Vous avez déjà rencontré des étiquettes. En effet, tous les contrôles (sauf l'étiquette) possèdent une zone de saisie pour la saisie de l'information et un libellé (du texte) éclairant l'utilisateur sur ce qu'il doit saisir. Le libellé est contenu dans un contrôle *Étiquette*.

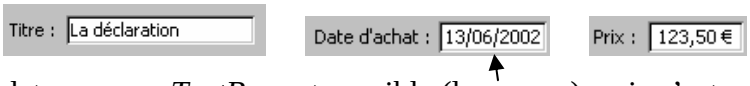
Voici trois exemples d'étiquettes, liées à un contrôle ou non.



1 E. Zone de texte (*TextBox*)

Le contrôle *Zone de texte* permet d'afficher ou de saisir des données de tout type (texte, date, chiffre). Enfin, plus précisément, le contenu d'une zone de texte est toujours du texte qu'il faut éventuellement convertir dans le type souhaité (entier, réel…) pour stocker la valeur saisie dans une variable.

Voici trois exemples de zones de texte, permettant respectivement de saisir du texte, une date et un montant monétaire.



Notez que saisir une date avec un *TextBox* est possible (la preuve) mais c'est une très mauvaise idée puisque ce contrôle autorise la frappe de n'importe quel caractère. Dans cette situation, il faut utiliser un *MaskEdBox*[1].

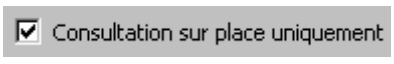
1 F. Bouton de commande (*CommandButton*)

Un *bouton de commande* est, avec la zone de texte, l'élément de base de toute interface graphique. Il permet de lancer l'exécution d'une procédure[2]. Cet objet ne possède pas de membre remarquable. Nous l'avons déjà étudié lors de la création de notre première application.

1 G. Case à cocher (*CheckBox*)

La *case à cocher* doit être utilisée pour saisir une valeur booléenne autonome.

Voici un exemple permettant d'indiquer si un livre peut être emprunté ou ne peut qu'être consulté sur place :

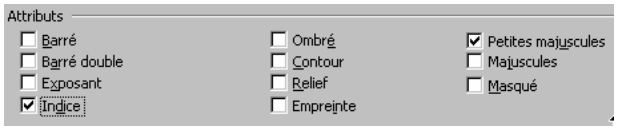


[1] Ce contrôle, que nous pourrions traduire en *boîte d'édition avec masque*, est référencé comme *Microsoft Masked Edit Control*… à ajouter avec *Projet/Composants…*

[2] Cette procédure est en fait la procédure événementielle liée à l'événement *Click* du contrôle *bouton*. Or, tout contrôle possède un événement *Click*. Cela signifie donc que tout contrôle peut exécuter du code lorsque vous cliquez dessus. Le bouton est-il donc inutile ? Absolument pas, au contraire : son rôle est *juste* de lancer du code. Il doit donc être utilisé lorsque vous voulez lancer du code sans rapport avec un autre contrôle.

Il est hors de question d'utiliser un autre contrôle, même si techniquement une zone de texte dans laquelle on saisirait *oui* ou *non* permettrait la saisie. La leçon à retenir, c'est que tout contrôle peut, si l'on se donne du mal, permettre de saisir tout type d'information. Mais, pour chacun de ces types, un seul contrôle est naturellement adapté (est fait pour cela).

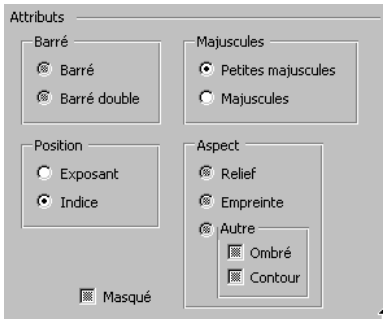
Autre exemple issu de Word (menu *Format/Police…*) :



Nous utilisons des cases à cocher (et non des boutons d'option) car les choix ne sont pas exclusifs : du texte peut être à la fois en indice et en petites majuscules.

Cela dit, certains choix sont exclusifs deux à deux : on ne peut pas être à la fois *Exposant* et *Italique* ou *Barré* et *Barré double*. Suis-je donc en train de renier tous mes principes ?

En appliquant strictement les règles d'emploi des contrôles, voici ce que l'on obtiendrait :



Que penser de cette présentation *new style* ? Ma foi, elle est nulle ! Certes, elle apporte beaucoup d'informations sur le format d'affichage. Le texte peut être :

- soit barré, soit barré double, soit pas barré du tout (les boutons d'option grisés dans un groupe indiquent qu'aucun choix n'est fait) ;
- soit en exposant, soit en indice, soit ni l'un ni l'autre ;
- soit normal, soit en petites majuscules, soit en majuscules ;
- soit en relief, soit en empreinte, soit ombré et/ou en contour.

Cela dit, ces informations sont d'un intérêt très limité et se paient au prix fort :

- la feuille a triplé de taille ;
- elle est beaucoup plus dense que la version originale ;
- j'ai dû utiliser des termes assez artificiels (position, aspect, autre) ou redondants (barré, majuscules) pour nommer les groupes ;
- le choix assez complexe présenté dans le groupe *Aspect* m'a obligé à inclure un groupe dans un autre et à mettre des cases à cocher. La sémantique est correcte : on peut être à la fois *Ombré* et *Contour*. mais pas si on est *Relief* ou *Empreinte*. C'est donc académiquement exact… mais très lourd.

Conclusion ? Eh bien, il faut respecter exactement les règles, y compris celle disant que toute règle se transgresse si l'on possède une bonne raison !

Si la fenêtre *Format/Police…* de Word ne suit pas les règles d'ergonomie habituelles, c'est parce que le faire n'apporterait rien, si ce n'est une feuille illisible.

1 H. Bouton d'option (*OptionButton*)

Les *boutons d'option* (on parle aussi de boutons radio) doivent être utilisés dans un groupe d'options (voir ci-dessous) pour permettre de sélectionner une option et une seule parmi plusieurs.

Le bouton d'option n'est donc pas fait pour saisir une valeur booléenne (cela, c'est la case à cocher), mais uniquement pour choisir une option. Or, qui dit choisir une option suppose d'avoir le choix entre plusieurs. C'est pourquoi ce bouton n'est jamais isolé, mais toujours en troupeau dans un groupe d'options.

1 I. Groupe et groupe d'options (*Frame*)

1 I 1. Définition

Le groupe est juste un conteneur permettant de regrouper de façon purement visuelle des contrôles. Il n'apporte pas de sémantique. Enfin… si vous placez des boutons d'option dans un groupe, ce dernier devient un groupe d'options, à savoir que, sans code supplémentaire de votre part, seul un bouton d'option à la fois pourra être sélectionné : le fait d'en sélectionner un désélectionne celui qui l'était déjà.

Attention, pour que le groupe joue son rôle de conteneur des contrôles qui sont dedans, vous devez dessiner le groupe puis lui ajouter les contrôles. Si vous vous contentez de glisser les contrôles dans le groupe, ils seront simplement superposés mais pas dedans.

Un groupe qui n'est pas un groupe d'option n'a d'intérêt que pour l'affichage, la structuration des contrôles. Il n'y a rien à en dire.

En revanche, il est important d'étudier le groupe d'options.

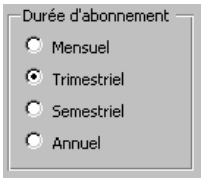
1 I 2. Utilisation raisonnée du groupe d'options

Nous venons de voir la théorie du groupe d'options. Comme c'est un contrôle complexe (enfin, très intuitif, mais un peu plus riche que la zone de texte par exemple), je vais me permettre quelques considérations générales pour vous permettre de l'utiliser dans les meilleures conditions.

Un *groupe d'options* contient des boutons d'option. À l'exécution, on ne peut sélectionner qu'une des options du groupe. On l'utilise donc pour saisir une valeur unique parmi une liste de valeurs prédéfinies. Pour des raisons d'ergonomie assez naturelles, ne mettez pas des dizaines de boutons d'option dans le groupe. Je vous conseille d'utiliser la liste modifiable lorsqu'il y a plus de 10 options.

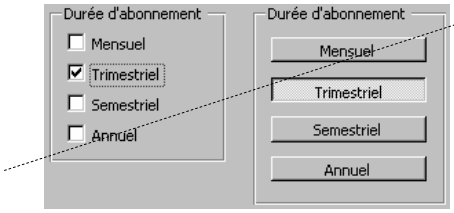
Exemple :

> Pour informatiser un club de sport proposant des abonnements de un, trois, six ou douze mois, vous utiliserez un groupe d'options :



J'ai dit que le groupe d'options contenait des boutons d'option. Vous devez en effet vous limiter à ce seul contrôle car son fonctionnement est normalisé en ce sens : une seule option peut être choisie.

Si vous utilisez un groupe tout court, vous pourrez utiliser n'importe quel autre contrôle à l'intérieur et obtenir ceci :



Non ! Ces contrôles ne sont pas utilisés dans leur sémantique habituelle.

Ainsi, ne faites jamais les types de groupe d'options ci-dessus. Le fait que vous les trouviez plus *sympas* n'est surtout pas un argument ! Le plus simple est donc d'utiliser systématiquement les groupes d'options. Comme vous n'avez pas le choix du contrôle contenu, vous éviterez toute erreur.

1 I 3. Gestion de la valeur du groupe d'option

Ce n'est pas encore fini. Ce contrôle possède une caractéristique assez particulière (partagée avec les zones de liste). L'astuce est la suivante : la valeur affichée (ou sélectionnée) dans le contrôle **n'est pas** la valeur du contrôle. Pour savoir quel bouton est sélectionné dans le groupe, il y a deux techniques :

- soit vous écrivez du code événementiel modifiant une variable lors du clic sur chaque bouton d'option. La valeur de cette variable vous indique alors le bouton sélectionné ;
- soit vous regroupez les boutons dans un groupe de contrôles que vous parcourrez comme un tableau pour connaître l'élément qui est actif.

Nous allons découvrir ces deux techniques. Ce qu'il faut dès maintenant accepter, c'est que le contrôle groupe ne dispose d'aucune propriété indiquant le bouton sélectionné, ni d'événement se déclenchant lorsqu'un bouton est cliqué. En clair, le groupe ne permet pas de déterminer lequel de ses éléments est sélectionné. Il faut accéder individuellement aux éléments pour le savoir.

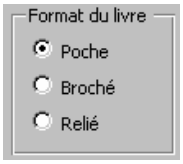
1er exemple : groupe d'options *Format du livre* (utilisation d'une variable)

Un livre peut être au format poche, relié ou broché. Ces valeurs sont exclusives : un livre broché n'est ni relié ni en poche. Le format le plus fréquent est le poche.

Lorsque je veux saisir le format d'un livre, quel contrôle vais-je utiliser ? Voyons :

- je dois choisir une valeur et une seule parmi plusieurs ;
- j'ai trois valeurs possibles et non cinquante.

Ces deux arguments plaident en faveur d'un groupe d'options (quelle surprise !) sélectionnant la valeur *Poche* par défaut. Le voici :



✎Exercice 26✎

Réalisez une feuille contenant ce groupe d'options. L'option *Poche* doit être sélectionnée par défaut (agissez sur sa propriété *Value*). La feuille doit aussi contenir un bouton. Lorsque l'on clique dessus, il doit afficher avec *MsgBox* le choix actuel (*Poche*, *Broché* ou *Relié*). Vous utiliserez une variable globale au module stockant le choix courant. Elle sera modifiée à chaque clic sur un des boutons et c'est elle que le bouton affichera. Comment faire pour que cette variable soit initialisée correctement à *Poche* ?

Comment régler notre problème au démarrage ? Il faudrait que la variable *FormatLivre* soit initialisée à *Poche* (pour être cohérent vis-à-vis du choix par défaut) au démarrage de l'application ou, au moins, avant que l'utilisateur ait eu l'opportunité de cliquer sur le bouton *Format*.

Il n'est pas possible d'initialiser une variable dès sa déclaration. Les lignes suivantes ne fonctionneront pas sous VB :

```
Dim FormatLivre = "Poche" As String
Dim FormatLivre as String = "Poche"
```

Quel drame, je ne marche pas !

Bah, moi non plus… et j'en fais pas toute une histoire.

On peut définir des constantes, mais une constante, cela est constant, n'est-ce pas, cela ne varie pas, donc cela ne nous servirait pas à grand-chose.

Nous allons chercher un événement qui se déclenche au démarrage de l'application. Il sera donc lié à la feuille.

Exercice 27

Cherchez, dans les différents événements liés à la feuille, celui qui pourrait régler notre problème et utilisez-le.

2e exemple : groupe d'options *Durée d'abonnement* (utilisation d'un groupe de contrôles)

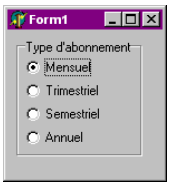
Nous souhaitons saisir la durée d'abonnement à un club de sport (l'abonnement peut être mensuel, trimestriel, semestriel ou annuel).

C'est une situation différente de l'exercice précédent. En effet, je veux obtenir de l'utilisateur une durée en mois (*1*, *3*, *6* ou *12* selon que l'abonnement est mensuel, trimestriel, semestriel ou annuel). La valeur qui nous intéresse n'est donc plus son libellé.

Nous pourrions reprendre la même technique que précédemment. Cela fonctionnerait. Cela dit, comme je souhaite récupérer des valeurs numériques, je peux faire plus simple.

Enfin, plus simple… la mise en œuvre sera plus lourde, mais, au final, le code sera plus léger.

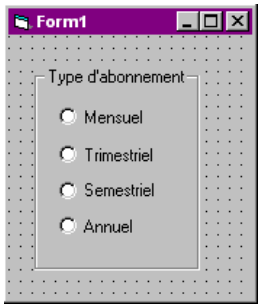
Voici le groupe que l'on veut réaliser je vous donne deux versions, selon que l'on affiche des nombres ou du texte (notre paramétrage ne dépendra pas du type d'affichage) :



Le principe va être de créer un groupe de contrôles. Attention à ne pas confondre ce groupe avec le groupe d'options :

- le groupe (*frame* sous VB) est un contrôle visuel servant de conteneur de contrôles. Si ces contrôles sont des boutons d'option, on parle de groupe d'options ;
- le groupe de contrôles est un concept algorithmique. Il revient à créer un tableau contenant des contrôles, cela permettant d'accéder aux différents contrôles grâce à un indice. (Et qui dit indice dit boucle !)

La première étape consiste à créer la feuille avec le groupe d'options :



Nous allons maintenant définir le groupe de contrôles. L'aide VB nous informe qu'un groupe de contrôles est, je cite, « [un] ensemble de contrôles qui partagent les mêmes nom,type et procédures d'événement. Chaque contrôle du groupe possède un numéro d'index qui lui est propre et permet de déterminer quel contrôle reconnaît un événement donné ».

L'idée est la suivante :

- nos quatre boutons d'option vont être regroupés dans l'équivalent d'un tableau appelé par exemple *ChoixDurée* ;
- je vais affecter un indice (numéro d'index) à chaque bouton d'options. Je pourrais utiliser 1, 2, 3 et 4, voire les nombres de 0 à 3 pour émuler la notion de tableau, mais je peux faire mieux : je vais affecter à chaque bouton sa durée en mois puisque c'est ce qui m'intéresse (ainsi, le bouton *Semestriel* aura comme indice 6) ;
- les événements qui s'appliquaient avant aux contrôles individuels s'appliquent maintenant au groupe, à charge au code de réagir selon le contrôle du groupe sur lequel l'événement a eu lieu. En fait, les contrôles individuels ne sont plus accessibles, seul le groupe est défini dans la fenêtre du code (liste modifiable de gauche). Bien entendu, la procédure événementielle aura comme paramètre l'indice du contrôle du groupe qui a subi l'événement.

✎Exercice 28✎

Eh bien, faites-le ! Je souhaite maintenant que le clic sur un des boutons d'option affiche avec un *MsgBox* la durée en mois de l'abonnement.

Conclusion sur le groupe d'options

Les valeurs proposées dans le groupe doivent être dans un ordre logique : du plus petit au plus grand, par ordre alphabétique ou par ordre d'usage :

- les durées d'abonnement étaient de la plus courte à la plus longue ;
- pour le format du livre, j'ai mis la valeur la plus fréquente (*Poche*) en premier. Les deux autres sont triées par ordre alphabétique.

Reprenons le groupe *Format du livre*. Vous remarquerez que les valeurs de chaque bouton d'option (pour l'affichage et le stockage) sont fournies *en dur*. Si je voulais gérer un nouveau format, par exemple *Livre électronique*, vous ne pourriez vous contenter de le rajouter dans *Format*. Il faudrait également ajouter un nouveau bouton d'option dans le groupe (ou, plus simplement, supprimer ce dernier et le recréer). Le groupe avec quatre options serait évidemment plus grand, ce qui risquerait d'obliger à repenser l'organisation de la feuille.

La mise à jour d'un groupe d'options représente un certain travail. Il ne faut donc utiliser ce contrôle que pour des choix stables. L'exemple typique est le titre de civilité d'une personne (M., M^{me} ou M^{lle}). Si les choix évoluent (par exemple, le choix d'un auteur pour un livre), on emploiera une zone de liste modifiable (voir le paragraphe 1.K).

De plus, le groupe est un contrôle volumineux et ne possède aucune aide à la sélection de l'option. Il ne faut donc pas l'employer pour choisir parmi 20 valeurs, mêmes stables : il occuperait tout l'écran et l'utilisateur passerait dix minutes à chercher la valeur qu'il souhaite. Dans ce cas, vous utiliserez la zone de liste modifiable ou la zone de liste *tout court*.

1 J. Zone de liste (*ListBox*)

Partez d'un groupe d'options. Enlevez les boutons d'option et ne gardez que leur libellé. Qu'obtenez vous ? Un groupe d'options sans petite pustule graphique sur laquelle cliquer.

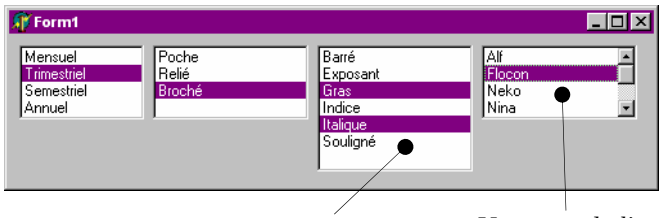
Et bien cela, c'est une zone de liste. Enfin, en vrai, la zone de liste possède d'autres caractéristiques qui la distinguent du groupe d'options : *Columns* (pour indiquer sur combien de colonnes on affiche les données), *List* (contenant les données affichées), *ListCount* donnant le nombre d'éléments dans la liste et *ListIndex* indiquant l'indice de l'élément sélectionné (– 1 s'il n'y en a pas, le premier élément est à l'indice 0 donc le dernier à l'indice *ListCount – 1*).

Pour remplir la zone de liste, vous pouvez :

- lors de la conception, remplir la propriété *List*. Pour cela, vous cliquez sur la petite flèche en regard de *(Liste)* et vous mettez une valeur par ligne dans la liste qui s'ouvre. Pour changer de ligne, ne faites pas *Enter*, qui terminerait la saisie, mais CTRL+*Enter* ;
- à l'exécution, modifier cette propriété *List* (voir les méthodes *AddItem*, *RemoveItem* et *Clear*).

Voyons quelques exemples :

Déjà, il n'y a pas de propriété Caption. *Si vous voulez mettre un libellé, il faut l'ajouter.*



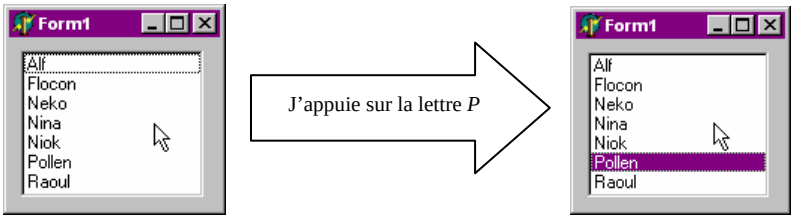
La zone de liste permet de sélectionner plusieurs éléments si on la configure pour cela (propriété MultiSelect *à* simple *ou* extended*).*

Une zone de liste possédera un ascenseur si sa hauteur n'est pas suffisante pour afficher tous ses éléments.

Est-ce tout au rayon des nouveautés ? Non ! Si vous voulez que les différents choix soient triés, inutile de le faire vous-même ! Tapez-les dans un ordre quelconque et définissez la propriété *Sorted*[1] de la zone de liste à *true*. Le contenu de la liste (soit la valeur de sa propriété *List*) sera automatiquement trié. C'est bien sympathique lorsqu'il y a beaucoup d'options !

Mieux encore, lorsque l'utilisateur doit choisir une valeur, il n'est pas obligé de faire défiler tout l'ascenseur : s'il appuie sur une touche, la sélection se positionnera automatiquement sur la première option commençant par le caractère correspondant. Ineffable si la liste est longue !

Démonstration :



[1] *Sorted* signifie *trié.*

Ces caractéristiques permettent de choisir entre le groupe d'options et la zone de liste : ce sera le groupe s'il y a peu d'options, la liste s'il y en a beaucoup.

Une dernière remarque :

> *ListIndex* indique le numéro de l'option sélectionnée. Que se passe-t-il si la zone de liste permet de sélectionner plusieurs éléments ? Vous trouverez la façon de s'en sortir dans l'aide.

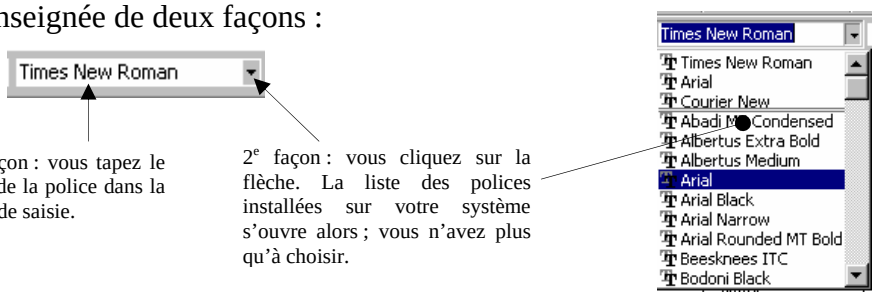
1 K. Zone de liste modifiable (*ComboBox*)

1 K 1. Présentation

Une zone de liste modifiable[1] est au choix :

- une zone de texte munie d'une flèche ouvrant une zone de liste ;
- une zone de liste (qui peut être masquée) associée à une zone de texte.

Prenons l'exemple de la zone de liste modifiable permettant de choisir une police sous Word. Elle peut être renseignée de deux façons :



Au final, ce qui distingue la liste de la liste modifiable tient en deux points :

- comme la liste ne possède pas de zone de saisie, le choix est limité aux valeurs proposées. Au contraire, la liste modifiable permet de choisir une option dans la liste ou d'en saisir une nouvelle si son paramétrage l'autorise ;
- la liste est par définition toujours déroulée. Elle occupe donc beaucoup plus de place mais affiche spontanément ses valeurs, ce qui permet une saisie plus rapide.

1 K 2. Manipulation

Il ne s'agit pas ici de la manipulation faite par l'utilisateur du programme, mais celle faite par vous, développeur.

Nous allons voir comment gérer les différents paramétrages que ce contrôle possède. Comme d'habitude, je ne peux que vous renvoyer vers l'aide pour plus d'informations.

Propriétés de base

La zone de liste modifiable est avant tout une zone de liste. Tout cela pour dire que nous retrouvons les propriétés *List*, *ListIndex* et *Sorted* dans leur rôle habituel.

Une propriété propre à la zone de liste modifiable est *Style*. Vous verrez dans l'aide que la valeur de cette propriété détermine le style (bah oui) de la liste, à savoir :

- si l'on peut ou non saisir une valeur dans la zone de texte autre qu'une valeur présente dans la liste ;
- si la liste se referme ou reste constamment ouverte.

[1] Certains emploient à tort le nom zone de liste *déroulante*. Certes, elle se déroule, mais ce n'est pas sa fonction !

Par défaut, *style* vaut *0 – Dropdown Combo*. On peut donc saisir la valeur dans la zone de texte ou la sélectionner dans la liste.

Propriétés avancées

L'utilisation de la zone de texte peut produire deux situations :

- l'utilisateur a tapé une valeur qui était présente dans la liste ;
- l'utilisateur a tapé une valeur qui n'est pas présente dans la liste. Et là, nous avons un problème.

Enfin, un problème… tout dépend de ce que nous voulons. La zone de liste peut avoir plusieurs comportements différents selon son paramétrage. On peut souhaiter que :

- l'utilisateur n'ait pas le droit de saisir une valeur qui n'est pas présente dans la liste (on parle de choix fermés). Dans ce cas, on modifiera simplement la valeur de *Style* ;
- l'utilisateur peut saisir une valeur qui n'est pas présente dans la liste (choix ouvert) ;
- l'utilisateur peut saisir une valeur qui n'est pas présente dans la liste et, dans ce cas, la valeur qu'il a saisie est automatiquement ajoutée à la liste.

Nous allons voir comment mettre en œuvre ces trois configurations.

✎Exercice 29✎

Dans une feuille, placez une zone de liste modifiable. Mettez trois valeurs dans *List* (sans originalité *Valeur 1*, *Valeur 2* et *Valeur 3*). Ne paramétrez aucune autre propriété de la liste. Ajoutez un bouton à la feuille pour afficher, lorsque l'on clique dessus, le numéro de l'élément sélectionné dans la liste (c'est toujours la propriété *ListIndex*).

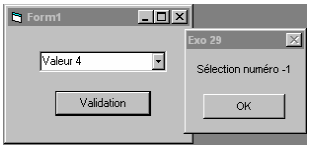
Que vaut *ListIndex* lorsque vous saisissez un élément présent dans la liste ? Et si cet élément n'est pas présent ?

Lorsque l'élément n'est pas présent, est-il ajouté automatiquement ?

Programmation

Nous allons maintenant programmer la zone de liste modifiable pour régler les problèmes soulevés dans la correction de l'exercice.

Reprenons le cas de la saisie d'une valeur qui n'était pas dans la liste :



Il est impossible de connaître la valeur saisie en passant par *List* puisqu'elle n'y figure pas. Eh bien, dans ce cas, il suffit d'utiliser la propriété *Text* de la zone de liste. Cette propriété donne le contenu de la zone de texte associée à la liste.

Maintenant, comment ajouter la valeur tapée pour qu'elle y figure à la prochaine ouverture de la liste ? Ce n'est pas très difficile : la propriété *List* de la liste est un objet possédant notamment la méthode *AddItem* pour lui ajouter une chaîne.

✎Exercice 30✎

Programmez la liste précédente pour qu'elle ajoute automatiquement la valeur saisie si elle n'est pas dans la liste. Vous avez en somme trois choses à faire :

1. Identifier le bon moment pour faire cet ajout (en clair, trouver l'événement adéquat).
2. Récupérer la valeur saisie (facile).
3. L'ajouter à *List* (pas difficile).

2. Les feuilles

2 A. Introduction

La première partie de cette séquence était réservée aux contrôles dont la finalité était la saisie ou l'affichage de valeurs, chacun étant spécialisé dans un type de données.

Nous étions tellement concentrés sur ces contrôles que nous n'avons pas pris le temps d'étudier le plus évident d'entre eux, à savoir la feuille, servant de contenant aux différents contrôles.

Nous allons maintenant voir comment faire pour gérer les feuilles. Nous aborderons trois points :

- comment créer ou supprimer des feuilles ;
- comment ouvrir ou fermer une feuille ;
- comment communiquer d'une feuille à une autre.

En fait, nous allons aborder des applications possédant plus d'une feuille.

2 B. Ajouter une feuille au projet

2 B 1. Comment faire

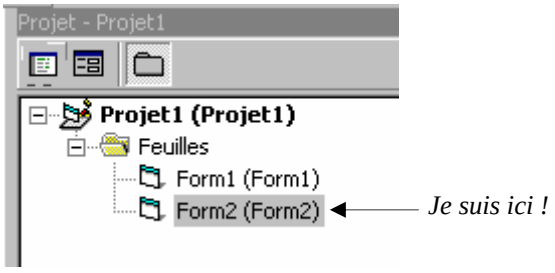
Alors là, c'est tout simple : il suffit de cliquer sur l'icone *Ajouter une feuille* de la barre d'outils ou de lancer la commande *Projet/Ajouter une feuille*.

Vous obtenez une nouvelle feuille. Attention, dès que vous manipulez plus d'une feuille, faites l'effort de les nommer plus clairement que *Form1* ou *Form2*.

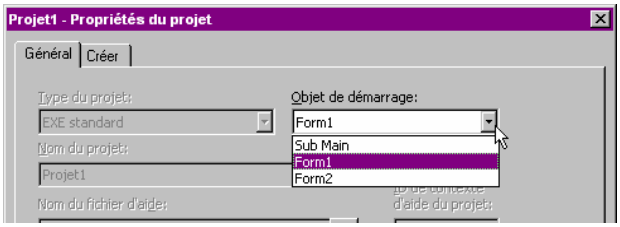
2 B 2. Les changements apportés

Voyons ce que le fait d'ajouter une feuille a apporté comme changement dans l'ensemble du projet. Déjà, vous avez une nouvelle feuille sous les yeux. Mais bon, ce n'est pas le seul changement.

La feuille est présente dans la fenêtre *Projet* :



De plus, dans la boîte de dialogue de la commande *Projet/Propriété de Projet1…*, on peut choisir l'une des deux feuilles comme objet de démarrage :



✎Exercice 31✎

Exécutez votre application avec les deux feuilles (comme elles sont vides, cela ira vite). Accédez à la seconde feuille… si vous y arrivez. Conclusion ?

La conclusion de l'exercice, c'est que créer une feuille n'a rien à voir avec le fait de l'afficher[1]. En fait, la création correspond à la réservation de la mémoire et à l'initialisation des différentes propriétés.

2 C. Ouvrir ou fermer une feuille

Nous venons de voir que créer une feuille ne voulait pas dire l'afficher à l'écran (le terme technique est *ouvrir*).

De même, détruire une feuille (à savoir libérer toute la mémoire qui avait été réservée lors de la création) n'a rien à voir avec le fait de le fermer.

♥ En fait, *ouvrir* et *fermer* sont des termes techniques. Leur équivalent en français courant est
♥ *montrer* et *cacher*.

♥ Pour ouvrir une feuille, on fait appel à sa méthode *Show*. Pour la fermer, ce sera sa méthode
♥ *Hide*.

Notez que, de même que nous ne créons pas les feuilles (c'est l'application qui s'en charge), nous ne les détruisons pas explicitement. C'est également l'application qui le fera lorsqu'elle se terminera. Enfin, il est techniquement possible de le faire nous même, notamment si nous créons des feuilles dynamiquement dans l'application. Mais c'est hors sujet ici.

✎Exercice 32✎

Dans la feuille *Form1*, placez un bouton qui, quand on appuie dessus, ouvre *Form2*. Dans *Form2*, placez un bouton qui, quand on clique dessus, ferme *Form2*.
Si vous avez un problème… essayez de le résoudre !

2 D. Communiquer des informations d'une feuille à une autre

✎Exercice 33✎

Je vous demande d'ajouter deux *TextBox*, un dans chaque feuille (appelé *Saisi* dans *Form1* et *Devine* dans *Form2*). L'idée est la suivante : lorsque je saisis un mot dans *Saisi*, ce mot doit être copié dans *Devine*. Ainsi, lorsque le clique sur *VoirForm2*, j'aurai l'impression que la seconde feuille a deviné le mot[2].

Bon, finalement, il n'y a pas de difficulté particulière, tout reste assez naturel.

Je vais néanmoins préciser et compléter ce que j'ai dit dans la correction. En effet, j'ai dit :

- lorsque vous manipulez une feuille (et ses contrôles) dans sa propre partie code, il est inutile de préfixer ses membres (méthodes ou contrôles) par son nom ;
- lorsque l'on manipule, dans le code d'une feuille ou dans un module, une autre feuille, il faut préfixer les membres de cette feuille par son nom.

[1] Ce qui est finalement logique : il ne serait pas raisonnable d'avoir dès le lancement d'une application toutes ses feuilles et boîtes de dialogue ouvertes !

[2] J'ai bien conscience que cela ne va impressionner personne de plus de 3 ans. Et encore, avec les jeunes de maintenant…

Ce n'est pas forcément très évident pour vous car on comprend mieux cela avec une pratique de la programmation objet qui n'est pas notre propos ici.

Je vais donc vous proposer une petite métaphore (c'est un peu ma spécialité). Je suis chez moi. Je demande à ma femme où est Nina, notre petit teckel sympa. Si elle me répond « dans le jardin », il n'y a pas d'ambiguïté : comme nous sommes chez nous, le *jardin* sans plus de précision est *notre* jardin. En revanche, si Nina a creusé un petit tunnel et s'ébat avec le westie du voisin, ma femme me répondrait « dans le jardin *du voisin* ».

Un autre exemple : si je suis dans la cuisine avec ma femme et qu'elle me demande d'allumer la lumière, je ferais preuve d'un esprit assez retord si j'allais allumer celle du perron. En effet, il est implicite que ma femme voulait plus de lumière dans la cuisine. En revanche, si nous sommes dans la cuisine et que nous attendons du monde, ma femme pourra me demander d'allumer la lumière *du perron.*

Concrètement, il n'est pas toujours évident de savoir quand préfixer par la feuille et quand ne pas le faire. Deux solutions s'offrent à vous :

1. Vous préfixez systématiquement par le nom de la feuille. C'est un peu lourd, mais cela fonctionnera. (De même que si, dans la cuisine, je demande à ma femme d'allumer la lumière de la cuisine, elle comprendra.)
2. Vous essayez d'optimiser votre code et tentez de ne préfixer par le nom de la feuille que quand c'est nécessaire. Dans ce cas, lorsque vous avez oublié de préfixer, vous aurez une erreur . Vous n'avez alors qu'à ajouter le nom de la feuille et tout ira bien.

Séquence 8

Les fichiers

Comment sauvegarder des données ? Nous allons le voir maintenant.

Contenu

Les différents types de fichiers :

- leur rôle ;
- leur syntaxe.

Capacités attendues

Utiliser tous les types de fichiers.

1. Introduction

Qu'avons-nous vu jusqu'à maintenant ? Les différentes instructions VB et la manipulation des contrôles.

Avec du temps et de l'expérience, vous pouvez écrire n'importe quel programme. Le fait que nous n'ayons fait ensemble que des petites applications un peu ridicules et courtes n'a aucune importance. Les concepts sont connus, reste à vous entraîner.

Cela dit, si vous écrivez une grosse application maintenant, vous allez être très frustrés. Pourquoi ? Prenons l'exemple de VB. Lorsque vous écrivez un programme, vous souhaitez bien entendu le sauvegarder pour pouvoir l'utiliser plusieurs fois. De même, sous Word, il y a des documents que vous tapez puis imprimez sans les sauvegarder (un courrier par exemple), mais d'autres que vous souhaitez conserver sur disque pour un usage ultérieur (votre CV…).

Lorsque vous utilisez une application quelconque (par exemple un programme que vous savez réalisé sous VB), c'est le même problème :

- certaines informations ne sont pas sauvegardées ;
- d'autres, au contraire, doivent l'être.

Par exemple, vous êtes entrepreneur et souhaitez gérer votre comptabilité sur ordinateur. À chaque décaissement (paiement d'une facture) ou encaissement, vous souhaitez mettre à jour votre comptabilité. Deux solutions s'offrent à vous :

- si vous avez la possibilité d'enregistrer votre comptabilité, l'application va reprendre le dernier solde qu'elle avait calculée, imputera votre décaissement ou encaissement et enregistrera le nouveau solde qui sera pris en compte à la prochaine utilisation du programme ;
- si rien n'est sauvegardé, l'application démarre sans aucune information. Vous devez alors saisir tous les mouvements comptables ayant eu lieu depuis la création de l'entreprise puis, enfin, saisir le nouveau décaissement ou encaissement. Et, comme vous ne pouvez rien sauvegarder, il faudra refaire tout cela lors du prochain mouvement. Ce n'est évidemment pas une solution acceptable.

En fait, la majorité des programmes exploitent des informations permanentes, c'est-à-dire stockées dans un fichier. Le fichier est utilisé en lecture pour récupérer des données d'initialisation puis en écriture pour sauvegarder les données finales.

Cette séquence va nous apprendre à manipuler les fichiers avec VB. Nous pourrons donc stocker des informations qui demeureront même après fermeture de l'application ou extinction de l'ordinateur.

2. Ce qu'est un fichier

2 A. Définition intemporelle

J'aurais tendance à vous dire que tout est fichier. En effet, un fichier est un ensemble d'informations cohérentes.

Pourquoi cohérentes ? Si je prends une partie de votre CV, le code source d'un programme VB et que je parsème le tout de quelques formules de calcul réalisées avec un tableur, j'ai un ensemble d'informations. Mais elles ne riment à rien.

Plus simplement, si je mélange un texte de chanson avec une lettre de réclamation à un fournisseur, j'obtiens également un résultat sans signification.

Un fichier contiendra donc des informations cohérentes. Par exemple, un fichier peut être :

- votre CV ;
- le texte d'une chanson ;
- une image ;
- un programme exécutable (*VB6.exe* par exemple) ;
- …

Toutes les informations stockées sur un disque le sont sous la forme de fichiers. Bref, sur votre disque dur, vous n'avez que des fichiers, qui peuvent contenir des programmes, des données ou autre.

Nous allons apprendre à manipuler les fichiers de données (un fichier programme ne se manipule pas, il s'exécute et c'est tout).

Nous verrons deux types de fichiers : les fichiers texte[1] et les fichiers binaires. Qu'est-ce que c'est que cela ? Poursuivez votre lecture et vous le saurez !

2 B. Fichiers obsolètes ou modernes ?

Depuis toujours, VB propose la routine *Open* (et d'autres associées) pour accéder de toutes les façons possibles aux fichiers. Cette méthode d'accès aux fichiers texte et binaires est très simple d'emploi et intuitive.

Pour suivre le développement du modèle objet, VB 6 propose une nouvelle façon d'accéder aux fichiers par l'intermédiaire d'objets créés pour l'occasion. C'est le modèle FSO pour *File System Object*.

Nul doute que ce modèle est promis à un bel avenir… en attendant, il est encore incomplet (il ne permet pas la création de fichiers binaires) et est trop riche pour nos maigres besoins.

Nous nous contenterons donc du vieux modèle : *Open* et ses amis. Si vous souhaitez découvrir FSO, à votre guise… l'aide en parle très bien !

2 C. Ce qui est commun à tous les fichiers : les descripteurs

2 C 1. Présentation

Que vous manipuliez un fichier texte ou binaire, il y a quelques principes incontournables que nous allons voir maintenant.

[1] Ce sont des fichiers de type texte, d'où *texte* et non *texte**s***.

Il faut bien distinguer le fichier physique du fichier logique. Que sont ces nouvelles notions ? Rien de terrible : le fichier physique, c'est le fichier qui est sur le disque. Le fichier logique est en quelque sorte la variable que VB utilise pour accéder au fichier physique.

VB utilise la notion de descripteur de fichier pour gérer le fichier logique. Ce terme barbare de descripteur correspond en fait juste à un entier identifiant le fichier physique sur lequel on travaille. Nous dirons donc plus simplement qu'il s'agit de numéros de fichiers car ce terme fait moins peur.

Par exemple, si j'accède à trois fichiers *client*, *facture* et *commande*, ils seront identifiés par les nombres *1*, *2* et *3* dans mon application. Ainsi, pour ajouter un client au fichier physique *client*, je l'ajouterai par programmation au fichier logique *1*.

Enfin, *1*, *2* ou *3* ou *27*, *19* et *42*… la seule contrainte est qu'un numéro ne soit associé qu'à un seul fichier.

Pour l'anecdote, notez que le numéro doit être compris entre 1 et 511 inclus.

2 C 2. Utilisation

Comme vous devez associer un numéro à tout fichier physique, il est raisonnable de déclarer une variable descripteur (numéro de fichier) entière pour chaque fichier.

Par exemple, je vais déclarer un numéro pour mon fichier *client* :

```
dim FichierClient as integer
```

Comprenez bien que *FichierClient* n'est pas le fichier lui-même, c'est juste une variable qui nous permettra d'y accéder.

Je vous ai dit que l'on pouvait associer n'importe quelle valeur entière aux numéros de fichiers, la seule contrainte étant de ne pas utiliser deux fois la même.

Au lieu d'affecter vous-même une valeur (par exemple *FichierClient = 4*), il est plus sage d'appeler la fonction *FreeFile* qui fournit à la demande des valeurs entières pas encore utilisées.

Pour initialiser le numéro, vous écrirez donc :

```
FichierClient = FreeFile
```

3. Les fichiers texte

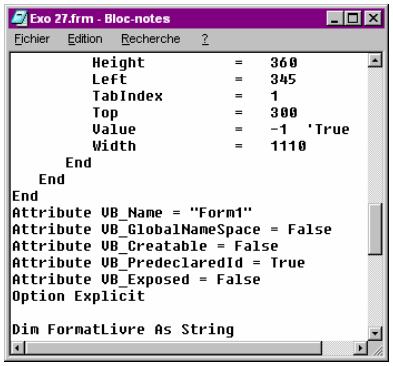
3 A. Définition

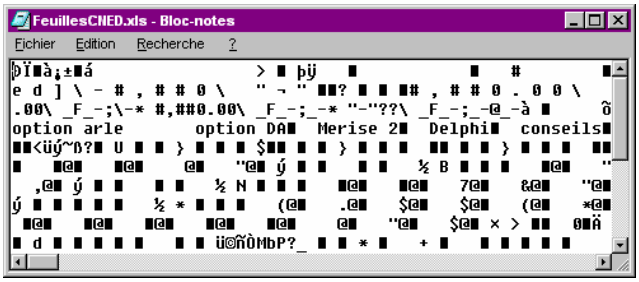
Un fichier texte, cela contient… du texte. En clair, une suite de caractères.

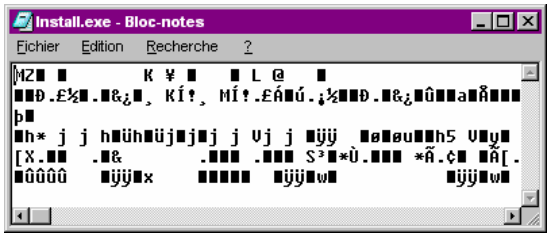
Attention, un document Word n'est pas un fichier texte, car il comprend non seulement les caractères constituant le texte, mais également, sous forme codée, les différentes mises en forme appliquées.

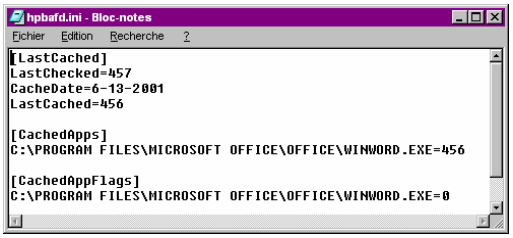
Pour savoir si un fichier contient du texte, essayez de l'ouvrir sous l'éditeur de texte *NotePad* (c'est un outil Windows accessible dans *Démarrer/Programmes/Accessoires/Bloc-notes*). Si le fichier qui s'ouvre est lisible, c'est un fichier texte. S'il affiche plein de choses étranges et que votre ordinateur bipe, ce n'est pas du texte[1].

Voici quelques exemples :









Nous voyons clairement que :

- le fichier *Exo 27.frm* est un fichier texte (c'est le source de la feuille de l'exercice crayon 27) ;
- le fichier *hpbafd.ini* est également un fichier texte (comme tous les fichiers *.ini*) ;
- en revanche, les exécutables (*install.exe*) et les classeurs Excel (*FeuillesCNED.xls*) ne sont pas des fichiers texte.

3 B. Utilisation du fichier texte

Nous allons nous en servir pour stocker du texte. Lequel ? Tout et n'importe quoi, cela dépendra de l'application.

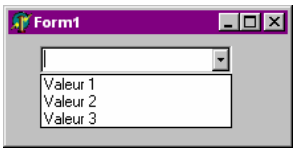
Soyons concrets avec un exemple. Retournez voir la correction de l'exercice 30 de la séquence précédente. Attention, n'allez pas voir mon support imprimé, mais le programme que vous avez dû taper.

[1] Attention, dans ce cas, ne modifiez surtout pas le fichier et quittez sans sauvegarder, sinon vous altéreriez de façon définitive le fichier.

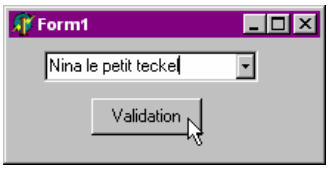
Ouvrez le programme sous VB puis exécutez-le en ajoutant dans la zone de liste une valeur quelconque. Vérifiez que la valeur est bien ajoutée. Quittez VB. Ouvrez de nouveau le programme et exécutez-le[1]. La valeur ajoutée est-elle encore présente ? Non.

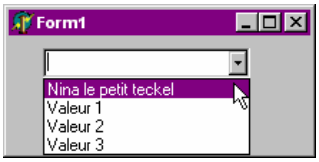
Démonstration :

J'exécute mon programme :

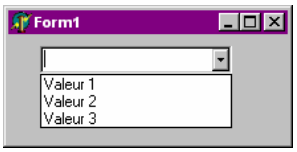


Je tape une valeur qui n'est pas dans la liste et je vérifie qu'elle est ajoutée :





Je quitte VB puis, je recharge et re-exécute le programme… *Nina le petit teckel* n'est plus dans la liste :



En fait, il n'y a rien d'anormal à cela : les valeurs *Valeur 1*, *Valeur 2* et *Valeur 3* sont toujours présentes car elles sont définies dans le source du programme. En revanche, *Nina le petit teckel* est une donnée qui n'a été stockée qu'en mémoire lorsque je l'ai ajoutée à *List*. Quand l'application se termine, la mémoire qu'elle utilisait est libérée. Notre valeur disparaît donc[2].

Une application du fichier texte pourrait être de stocker le contenu de la propriété *List* (soit les différentes chaînes) dans un fichier texte lorsque l'on quitte l'application et, au démarrage, d'aller lire les chaînes stockées pour les placer dans *List*. Ainsi, chaque chaîne ajoutée resterait dans la zone de liste pour les utilisations futures.

C'est un tellement bel exemple que nous le traiterons dans le TD à venir.

3 C. Le fichier texte sous VB

Fixons-nous un objectif : écrire un programme **console** prenant les en-têtes de tous les sous-programmes d'un module VB donné et les copiant dans un fichier texte. Le fichier contiendra d'abord les en-têtes des procédures puis celles des fonctions. Pourquoi un programme *console* ? Les fichiers textes sont utilisables dans tous les modes (console ou graphique). Si nous nous plaçons en mode console, c'est pour nous concentrer sur les fichiers sans avoir à se soucier de problèmes de contrôles.

[1] Ou lancez deux fois de suite le programme exécutable créé par VB.
[2] En pratique, elle n'est pas effacée, mais c'est tout comme car son contenu a perdu toute signification.

3 C 1. Déclaration et initialisation

Déclarer un fichier revient à créer une variable descripteur chargée de stocker son numéro de fichier. Nous l'avons fait dans le paragraphe 2C.

Nous affectons ensuite au descripteur une nouvelle valeur avec *FreeFile*.

3 C 2. Association

Déclarer une variable est une chose, l'initialiser est autre chose. Les variables fichier n'échappent pas à la règle. Le problème, c'est de savoir ce que signifie initialiser un fichier.

En fait, c'est naturel : notre objectif est de manipuler un fichier présent sur le disque. Avoir initialisé le descripteur avec *FreeFile* lui a juste attribué un numéro unique. Aucune association n'a encore été faite entre notre descripteur et un fichier physique (réel) présent sur un disque.

C'est le rôle de l'instruction *Open*, à qui on fournira un descripteur **initialisé** (par *FreeFile*) et un chemin d'accès de fichier physique.

Voici sa syntaxe pour le moment[1] :

```
Open CheminAccès as #Descripteur
```

Voir ci-dessous pour comprendre ce que le petit # vient faire.

Voici un exemple :

```
Dim JeNaiPasIntérêtÀLeModifierCeluiLà as integer
Open "C:\autoexec.bat" as #JeNaiPasIntérêtÀLeModifierCeluiLà
```

Que vient faire ce caractère # entre *as* et le descripteur ? Mon dieu qu'il est étrange, ce petit dièse ! En fait, c'est une abréviation américaine : là où nous abrégeons « numéro 4 » en « n° 4 », les américains abrègent en « # 4 ». Bref, le dièse est synonyme de numéro.

Or, je vous rappelle que le descripteur est un numéro de fichier. Il faut donc lire l'instruction *Open* ci-dessus comme « ouvrir le fichier *CheminAccès* sous le numéro *Descripteur* ».

Je vous avoue que le dièse est facultatif : si vous ne le mettez pas, VB comprendra tout autant votre instruction. Je l'avais donc directement omis dans la première version de ce cours.

Le problème, c'est que certaines des instructions qui suivent nécessitent le # qui n'est plus facultatif. Prenons l'exemple de l'instruction *Print #* permettant d'écrire dans un fichier. Le dièse est obligatoire. Pourquoi ? Parce que si vous l'omettez, vous faites appel à l'instruction *Print tout court* qui existe et écrit dans la feuille courante, ce qui n'a rien à voir !

Ainsi, j'avais deux possibilités :

- ne parler du # que lorsqu'il est obligatoire. Mais alors, vous vous mélangeriez les pinceaux et ne sauriez plus quand l'utiliser et quand l'omettre ;
- en parler pour chaque instruction, qu'il soit facultatif ou obligatoire. Ainsi, pas de question à se poser : on met le dièse et c'est tout !

Open permet en fait :

- de faire le lien entre le descripteur et le fichier physique ;
- de définir le mode d'accès au fichier : ajouterons-nous des données, nous contenterons-nous de les lire…

Vous devrez donc fournir trois paramètres à *Open* :

- un descripteur ;
- le chemin d'accès d'un fichier ;

[1] J'insiste : pour le moment. En effet, la syntaxe est incomplète car *Open* contient aussi le mode d'ouverture.

– le mode d'ouverture.

Nous allons voir tout cela.

3 C 3. Modes d'ouverture du fichier

L'assignation permet de lier un fichier logique à un fichier physique. Concrètement, nous n'avons pas encore accédé au fichier.

Vous connaissez les quatre modes d'ouverture de fichier :

– en création (le fichier est créé ; s'il existait déjà, il est supprimé puis créé). Vous pouvez alors le remplir ;
– en ajout (on peut juste ajouter des données à la fin du fichier) ;
– en lecture (on peut lire le contenu du fichier mais pas le modifier) ;
– en lecture/écriture (on peut lire ou écrire dans le fichier).

Ces différents modes seront autant de paramètres à fournir à *Open*. Voici la nouvelle syntaxe :

```
Open CheminAccès for Mode as #Descripteur
```

Sous VB, les mots clé correspondant aux différents modes d'ouverture sont les suivants :

Ouverture d'un fichier texte	
MODE D'OUVERTURE	MOT CLE VB
création	Output
lecture	Input
ajout	Append
lecture/écriture	*néant*

Voici un exemple d'ouverture permettant de lire mon fichier *autoexec.bat* :

```
Dim JeNaiPasIntérêtÀLeModifierCeluiLà as integer
Open "C:\autoexec.bat" for Input as #JeNaiPasIntérêtÀLeModifierCeluiLà
```

Au fait, en anglais, *Input* correspond à *entrée* et *Output* à *sortie*. Or, quand vous ouvrez un fichier en *Input*, c'est pour extraire des données, donc pour les sortir du fichier. Incohérence ? Non, tout dépend du point de vue. En effet, récupérer des données venant du fichier, c'est sortir les données du fichier pour les récupérer dans l'application. Elles entrent donc dans l'application.

Le mode d'accès est donc vis-à-vis de l'application et non du fichier. (C'est un truc mnémotechnique pour ne pas se tromper de mode.)

Notez que l'on ne peut pas modifier une ligne dans le fichier (l'accès en lecture/écriture n'existe pas). La seule façon d'écrire dans le fichier texte, c'est d'ajouter des lignes à la fin.

Les fichiers binaires permettront la modification. Pourquoi cela n'est-il pas possible dans les fichiers textes ? Nous le verrons ci-dessous.

Comment passer d'un mode d'ouverture à l'autre ? Par exemple, nous sommes en mode lecture et nous voulons passer en mode ajout sur le même fichier. Comment faire ? Il suffit de fermer le fichier (nous allons voir comment) puis de le rouvrir dans le nouveau mode.

Je vous avoue qu'*Open* propose d'autres options en plus de ce que nous venons de voir. Pour les découvrir, un réflexe : *F1*. (Elles n'ont pas d'intérêt à notre niveau, raison pour laquelle je n'en parle pas.)

3 C 4. Fermeture du fichier

Quand vous avez fini de lire un livre, vous le fermez. Eh bien, lorsque vous avez fini d'accéder à un fichier, vous le fermerez également avec l'instruction *Close* :

```
Close #descripteur
```

En fait, toute manipulation d'un fichier suit le modèle suivant :

```
dim Numéro as integer

Numéro = FreeFile
Open … for … as #Numéro
…
// accès au fichier (voir ci-dessous)
…
Close #Numéro
```

En d'autres termes :

- on assigne le fichier physique au fichier logique et on ouvre le fichier (*Open*) ;
- on lit et/ou écrit dans le fichier (voir ci-dessous) ;
- on ferme le fichier (*Close*).

Ne pas fermer le fichier n'entraînera pas forcément un plantage, mais causera à coup sûr des problèmes : perte de données, consommation excessive de mémoire… Bref, n'oubliez pas !

3 C 5. Accès au contenu du fichier

Une fois le fichier ouvert, et avant de le fermer, nous allons bien entendu lire ou écrire dedans :

- pour lire, on utilisera les instructions *input #* ;
- pour écrire, ce sera *print #*.

Ces deux instructions manipulent des lignes entières (y compris donc le saut de ligne).

Leur syntaxe est la suivante :

```
input #descripteur, ChaîneLue
print #descripteur, ChaîneÉcrite
```

3 C 6. Exemple

Nous allons écrire un programme console créant un fichier texte *nombres* à la racine du disque dur et contenant les nombres de 1 à 10 000 (un nombre par ligne). L'intérêt de ce programme ? Il n'y en a aucun.

Le principe est le suivant :

- on assigne le fichier ;
- on l'ouvre en création (il est en effet supposé ne pas exister) ;
- on fait une boucle de 1 à 10 000, chaque valeur étant écrite dans le fichier ;
- on ferme le fichier.

Exercice 34

Faites ce programme.

3 C 7. Parcours d'un fichier

Un tableau possède un nombre d'éléments précis. Pour le parcourir du début à la fin, il suffit d'employer une boucle pour allant de *1* à *n* si le tableau contient *n* éléments.

Un fichier texte peut contenir 10 lignes, 200 lignes, un million de lignes ou éventuellement aucune[1]. Bref, on ne peut pas connaître sa taille. Pour le parcourir, on est obligé d'utiliser une boucle *tant que*. Le principe est simple : on dira « tant que l'on n'est pas en fin de fichier, on avance d'une ligne ».

Comment savoir si l'on est en fin de fichier ? En utilisant la fonction *Eof*[2], qui prend un fichier en paramètre et renvoie un booléen (*vrai* si l'on est en fin de fichier, *faux* sinon) :

```
fonction Eof (descripteur) : booléen
```

Aïe, cette fois, nous sommes coincés : *Eof* n'accepte pas le #. Eh oui…

Exercice 35

Complétez le programme précédent pour qu'après avoir fermé le fichier il l'ouvre de nouveau pour lire tous les éléments qu'il contient. Comme il n'est pas question de les afficher, insérez un compteur s'incrémentant à chaque lecture. Vous afficherez la valeur du compteur en fin de lecture (il doit valoir 10 000).

3 C 8. Réalisons notre objectif

Rappelons notre objectif : écrire un programme **console** prenant les en-têtes de tous les sous-programmes d'un fichier VB donné et les copiant dans un fichier texte. Le fichier contiendra d'abord les en-têtes des procédures puis celles des fonctions.

Inutile d'optimiser notre algorithme. Je vous propose le principe suivant :

- on ouvre le fichier contenant le source VB (*source*) et celui qui contiendra tous les en-têtes (*cible*) ;
- on parcourt le fichier *source* du début à la fin en lisant chaque ligne. Si la ligne contient le mot *sub*, on l'écrit dans *cible* ;
- on ferme *source*, puis on le rouvre ;
- on parcourt *source* du début à la fin en lisant chaque ligne. Si la ligne contient le mot *function*, on l'écrit dans *cible* ;
- on ferme les deux fichiers.

Exercice 36

Faites le programme. Je veux que l'utilisateur puisse rentrer les chemins de source et cible. Pour savoir si une ligne contient *sub* ou *function*, vous utiliserez la fonction VB *InStr* (voir l'aide).

Exercice 37

Juste en passant, un petit exercice… La correction de l'exercice précédent n'est pas très efficace car le traitement du balayage dans le fichier source est effectué deux fois de façon identique (seul le mot cherché change). Je vous demande d'optimiser ce programme en créant un sous-programme de balayage qui sera appelé deux fois. Prêtez une attention toute particulière aux paramètres. Je veux que tous les accès aux fichiers soient réalisés dans le nouveau sous-programme.

Vous aurez besoin de *Dir* (voir l'aide) pour un maximum d'efficacité.

[1] Si vous créez un fichier puis le fermez juste après, il sera vide.

[2] *Eof* pour *End Of File* (fin de fichier).

Une remarque importante :

Si nous avions réalisé les ouvertures et fermetures de fichier dans le programme principal, nous aurions évité l'astuce avec *Dir*. Je vous le déconseille pourtant formellement.

En effet, la bonne méthode consiste à ouvrir le fichier le plus tard possible et à le fermer le plus vite possible, même si cela revient à l'ouvrir et à le fermer dix fois dans un programme. Pourquoi ? Car en ne faisant pas cela, vous risquez de ne pas savoir dans quel état est un fichier (ouvert ? fermé ?) si votre programme est un peu long.

4. Les fichiers binaires

4 A. Définition

Un fichier binaire contient des données stockées sous la forme binaire (comme en mémoire).

Si vous stockez la valeur 73 dans un fichier texte, c'est les caractères *7* et *3* qui seront écrits. En revanche, si vous stockez 73 dans un fichier binaire, c'est la valeur binaire associée à 73 qui sera stockée, soit 01001001[1].

Lorsque vous ouvrez le fichier sous *NotePad* (ou faites, pour les initiés, la commande *type* sous Dos), le fichier sera interprété comme un fichier contenant des caractères. Ainsi, l'application, trouvant la valeur 73 (enfin, 01001001) affichera le caractère ayant le caractère ASCII 73, soit le caractère *I.*[2].

Comme un fichier binaire contient les mêmes informations que celles stockées en mémoire, nous aurons le même problème : si l'on ne connaît pas le type des informations stockées, on ne peut pas les récupérer.

C'est pourquoi les fichiers binaires sont généralement des fichiers typés : ce ne sont pas des fichiers *tout court*, mais des fichiers de *quelque chose*. (Fichier d'entiers, fichier de structures client…)

À cette différence importante près, ils se manipuleront de la même façon (et avec les mêmes instructions) que les fichiers texte.

4 B. Utilisation du fichier binaire

Nous allons nous en servir pour stocker des données. Lesquelles ? Tout et n'importe quoi, cela dépendra de l'application.

Soyons concrets avec un exemple. Au début de ce chapitre, j'ai parlé d'un programme de comptabilité. Eh bien, au fur et à mesure qu'une opération comptable est saisie, elle sera stockée dans un fichier d'opérations comptables. Ainsi, à chaque démarrage de l'application, les anciennes opérations pourront être chargées et le programme reflétera l'état actuel de la trésorerie.

4 C. Le fichier binaire sous VB

4 C 1. Déclaration

Nous avons toujours besoin d'un descripteur.

4 C 2. Initialisation

Que le fichier soit texte ou binaire, il faut toujours associer le fichier logique (la variable) au fichier physique. C'est toujours *Open* qui s'y colle, mais là, vous aurez une syntaxe un peu différente car il y aura plus à dire.

[1] Voir séquence 2, paragraphe 1B1.

[2] Dans le paragraphe 3.1, je vous avais dit que votre ordinateur pouvait biper en affichant un caractère binaire. C'est pour une raison simple (enfin…) : le bip est un caractère de contrôle (non visuel) ayant le code ASCII 7. Ce qui veut dire que si vous affichez caractère de code ASCII 65, vous verrez un *A*. Si vous affichez celui de code 7, vous verrez… rien, mais vous entendrez un bip. Bref, les caractères en informatique sont un peu plus étendus que ceux dans la vraie vie. Claudel a écrit « l'œil écoute ». J'aurais pu lui répondre « le caractère parle ».

En effet, le fichier texte contient des lignes de texte et c'est tout, tandis que le fichier binaire contient des informations de même nature, tout comme un tableau[1].

Ainsi, un fichier binaire sera un fichier d'entiers, un fichier de clients (client étant un type structuré défini par l'utilisateur) voire un fichier de tableaux de 10 réels.

On voit qu'il existe un nombre infini de types de fichiers binaires puisque n'importe quel type de données peut être stocké dans un fichier : non seulement les types de base (entier, réel, caractère…) mais aussi les types définis par l'utilisateur (tableau, structures). La seule exception est le fichier : vous ne pouvez pas définir de fichier contenant des fichiers.

Le fait de ne pouvoir stocker que des éléments de même type semble contraignant. Nous verrons en 4C6 que cela permet l'accès direct.

Avec un fichier texte, l'instruction de lecture lisait tous les caractères d'une ligne. Elle s'arrêtait donc en arrivant sur un caractère de fin de ligne. L'écriture se faisait de la même façon : tous les caractères de la chaîne passée en paramètre étaient écrits.

Maintenant, c'est différent : la notion de fin de chaîne ou de ligne n'existe que pour les chaînes et les lignes. Comment faire ici ? Eh bien, comme tous les éléments sont de même type, ils occupent la même place en mémoire donc sont constitués du même nombre d'octets. Lors de l'ouverture du fichier avec *Open*, on précisera le nombre d'octets occupés par un élément du fichier. L'écriture ou la lecture se fera alors sur le nombre d'octet spécifié. Ainsi, si vous mettez une valeur qui ne correspond pas à la taille des éléments, le fichier sera totalement corrompu[2].

En clair, VB vous fait confiance quant à la taille des enregistrements. En fait, il n'a pas le choix : pour lui, un fichier n'est qu'une suite de bits (huit bits formant un octet), il n'en connaît pas le contenu. Si vous dites que les éléments font quatre octets, il lira ou écrira les octets par paquets de quatre. Si vous dites qu'ils font seize octets, il y accédera seize par seize.

Voici la nouvelle syntaxe d'*Open*.

```
Open CheminAccès for Mode as #Descripteur Len=taille
```

Taille est la taille en octets des éléments constituant le fichier. Cette information n'est d'ailleurs pas évidente à obtenir : si je déclare un type *Client* personnalisé, pouvez-vous me dire facilement le nombre d'octets qu'il occupe ? Je ne pense pas.

C'est pourquoi vous utiliserez avec profit l'instruction *Len* qui vous renvoie :

- le nombre de caractères de la chaîne passée en paramètre (cela ne nous intéresse pas ici) ;
- la taille en octets de la variable passée en paramètre.

Il n'y a pas d'ambiguïté entre ces deux résultats puisqu'un caractère occupe toujours un octet. Ainsi, le nombre de caractères d'une chaîne, c'est sa taille en octets.

Par exemple :

- *len (i)* vaut 2 si *i* est une variable entière ;
- *len (r)* vaut 4 si *r* est une variable réelle ;
- *len (Cl)* vaut… tout dépend de ce que vous avez mis dans le type *Client* (je suppose bien entendu que *Cl* est de type *Client*).

[1] D'ailleurs, tableau et fichier binaire sont deux concepts totalement identiques, l'un étant stocké en mémoire, l'autre sur disque.

[2] Par exemple, si vous manipulez un fichier de réels en indiquant comme taille d'éléments celle d'un entier, vous ne stockerez que la moitié du réel. Attention, pas la moitié au sens division par 2, mais juste les deux premiers octets des quatre que constituent un réel, puisqu'un entier est codé sur deux octets. Inversement, si vous lisez un fichier de réels comme si c'était un fichier d'entiers, vous aurez deux fois plus de données car *un réel = 4 octets = 2 * 2 octets = 2 entiers*.

En résumé, *len* est très simple à utiliser et fournir vous-même une taille en octets ne peut que mener à la catastrophe. Je souhaite donc tellement que vous utilisiez systématiquement *len* que je vais l'intégrer dans la nouvelle syntaxe d'*Open*, qui devient :

```
Open CheminAccès for Mode as #Descripteur Len=Len(variable)
```

Attention, n'oubliez pas que le paramètre de *len* est une variable de même type que les éléments du fichier.

4 C 3. Chaîne et fichier binaire

Attention à une grosse astuce : une chaîne de caractères (type *string*) contient un nombre de caractères variables, ce qui n'est pas compatible avec notre obligation d'un type qui possède toujours la même taille.

Vous ne pourrez donc pas mettre des données de type chaîne, mais uniquement des chaînes de longueur fixe. Qu'est cela ?

Ceci :

```
dim Ch as string*30
```

Ch est une variable chaîne de caractères de longueur 30, donc ne pouvant contenir au maximum que 30 caractères.

Tous vos types structurés devront contenir des chaînes de longueur fixe, la longueur étant quelconque (j'ai mis 30, on peut évidemment mettre une autre valeur).

4 C 4. Modes d'ouverture du fichier

Nous retrouvons nos modes *Input*, *Output* et *Append*, mais aussi un petit nouveau, *Random*, qui permettra de se positionner n'importe où dans le fichier pour lire ou modifier un enregistrement. Je vous conseille de vous servir systématiquement de *Random* car il permet de tout faire.

4 C 5. Fermeture du fichier

Rien n'a changé, c'est toujours *Close*.

4 C 6. Accès direct

Pour le moment, le fichier binaire manque d'intérêt par rapport au fichier texte. En effet, au prix de quelques petites manipulations, nous pouvons convertir toute donnée sous la forme d'une chaîne de caractères et ainsi la stocker dans un fichier texte.

En fait, le fichier binaire possède une caractéristique essentielle : il permet l'accès direct.

Pour accéder au contenu d'un fichier texte, la seule façon est de l'ouvrir puis de le lire en partant du début. Si nous voulons la dernière ligne d'un fichier qui en contient un million, nous sommes obligés de lire les neuf cent quatre-vingt-dix-neuf premières, qui ne nous intéressent pas, avant d'accéder à la dernière.

Pourquoi est-ce ainsi ? C'est le corollaire de la souplesse du fichier texte. Il permet de stocker des éléments ayant une taille variable. En effet, un fichier texte est en fait un fichier contenant des lignes. Ces lignes sont de taille variable (sur une ligne, je peux avoir 5 caractères et sur la suivante, 200). L'application n'a donc aucun moyen de savoir où se situe la ligne cherchée dans le fichier. Il doit donc toutes les parcourir.

Une métaphore ?

Eh bien, imaginions que vous cherchiez une maison dans une rue, en ne connaissant que son numéro. Impossible de se dire « bon, je suis au numéro 12, je cherche le numéro 715, je vais donc avancer de 823,45 mètres et je serai rendu ». En effet, les maisons et les terrains ayant des tailles variables, vous ne pouvez pas connaître la distance entre deux maisons[1]. Vous êtes donc obligé de parcourir toute la rue jusqu'à arriver devant le 715 cherché (et alors vous apercevoir qu'il n'y a personne…).

Dans un fichier binaire, c'est le contraire. Toutes les données stockées sont de même type donc occupent la même place en mémoire (donc sur le disque). Ainsi, connaissant la position du premier élément du fichier, il suffit de réaliser une opération mathématique simple pour savoir où se trouve l'élément numéro 56 (ou 34, ou 567…).

Une métaphore ?

En fait, c'est exactement ce qui se passe avec les tableaux. Les fichiers binaires sont d'ailleurs l'équivalent des tableaux, ceux-ci étant en mémoire et ceux-là sur disque ; dans les deux cas, les éléments sont stockés de façon contiguë.

Comme un tableau ne contient que des éléments de même type, chacun occupe la même place en mémoire. Il est donc très simple d'accéder directement à l'élément que l'on souhaite[2].

Comme fichiers binaires et tableaux sont similaires (voire équivalents), la notion d'indice existe pour les deux notions. C'est très utile lorsque l'on sait que pour lire ou écrire un élément, il faut être positionné dessus (donc que l'indice corresponde au numéro de l'élément).

Nous allons voir que les instructions permettant de lire ou d'écrire dans un fichier permettent en même temps de se positionner au bon endroit.

4 C 7. Accès au contenu du fichier

Dans un fichier texte, la notion de ligne est très importante. (C'est d'ailleurs la différence entre un fichier texte et un fichier de chaînes, ce dernier étant un fichier binaire.)

Dans le cours que vous êtes en train de lire, j'ai écrit des phrases longues, courtes, je vais à la ligne… tout cela pour que la présentation du texte vous aide à l'apprentissage.

Dans un fichier binaire, cela n'a plus de sens : on écrit des données et c'est tout. Elles sont écrites les unes à la suite des autres. Si cela ne vous semble pas clair, disons que le fichier binaire ne contient qu'une ligne… qui peut être très longue.

Bref, les instructions *Input #* et *Print #* ne sont plus adaptées. Nous allons utiliser *Put #* et *Get #* qui vont permettre de dire de lire ou d'écrire n'importe où dans le fichier car on peut indiquer le numéro de l'élément à lire ou à écrire (donc à modifier) :

```
Put #descripteur, Numéro, VariableÉcrite
Get #descripteur, Numéro, VariableLue
```

[1] Sauf dans le cas où les maisons sont numérotées en système métrique, le numéro de la maison correspondant à la distance en mètres séparant la maison du début de la rue. J'ai habité 3 ans dans une telle rue et il m'en a fallu 2 pour comprendre cela et ainsi ne plus être choqué par le fait que j'habitais au 319 et mon voisin au 326 alors que nos maisons étaient mitoyennes.

[2] Supposons un tableau d'entiers. Un entier occupe 2 octets en mémoire. Dans ce cas, si le premier débute à l'octet 456, il se termine à l'octet 457. Le deuxième élément du tableau sera donc dans les octets 458 et 459, le troisième en 460 et 461 et le i^e en *(456+(i–1)*2)* et *(457+(i–1)*2)*.

Le premier élément d'un fichier possède le numéro 1. Notez bien que si nous manipulons un fichier d'entiers, les données lues ou écrites ne peuvent être que des entiers. Et idem pour tout type. La variable lue ou écrite sera donc d'un type cohérent avec le contenu du fichier.

Seconde version, on n'indique pas de position. L'accès se produira là où l'on est :

```
Put #descripteur,, ChaîneÉcrite
Get #descripteur,, ChaîneLue
```

Attention, les deux virgules qui se suivent ne sont pas une erreur : elles permettent d'indiquer au compilateur l'absence du deuxième paramètre.

4 C 8. Exemple

Nous allons écrire un programme console créant un fichier binaire *nombres* à la racine du disque dur et contenant les nombres entiers de 1 à 10 000. L'intérêt de ce programme ? Toujours aucun.

Exercice 38

Faites ce programme. (C'est l'exercice 34, sauf que cette fois nous manipulons un fichier binaire et non plus texte.)

4 C 9. Parcours d'un fichier

Eof est plus que jamais d'actualité !

```
fonction Eof (descripteur) : booléen
```

Exercice 39

Complétez le programme précédent pour qu'après avoir fermé le fichier il l'ouvre de nouveau pour afficher son 5 555^{e} élément.

4 C 10. Nombre d'éléments d'un fichier

Il n'existe pas de fonction renvoyant directement le nombre d'éléments d'un fichier. Pour obtenir cette information, nous utiliserons *Lof* (pour *Length Of File*) renvoyant la longueur en octet du fichier. Pour avoir le nombre d'éléments, nous diviserons le résultat de *Lof* par la taille d'un élément (obtenu avec *Len*).

Syntaxe de Lof :

```
fonction Lof (descripteur) : integer[1]
```

Pour connaître le nombre d'éléments stockés, on écrira *Lof (descripteur)/Len (Variable), variable* étant une variable du même type que les éléments du fichier.

Ce résultat est forcément un nombre entier si vous n'avez pas fait d'erreur.

4 C 11. Accès direct

Les fichiers binaires proposent des instructions et fonctions permettant :

- de connaître l'indice courant du fichier ;
- de se déplacer dans le fichier (en modifiant l'indice).

[1] Enfin, très précisément, ce n'est pas *integer* mais *long*, type entier codé sur 4 octets donc n'étant pas limité à la valeur 32 767 comme l'*integer*.

Ce qui est amusant, c'est que c'est la même instruction, *seek* :

- la fonction *seek* renvoie la position courante dans le fichier ;
- la procédure *seek* permet de se positionner dans le fichier.

Je ne détaille pas *seek*. Cette instruction est généralement inutile car *Put* et *Get* permettent de spécifier la position.

✎Exercice 40✎

Nous allons mettre tout cela en œuvre. Définissez un type *Client* (nom, adresse et téléphone). Stockez dans un fichier trois clients que vous aurez saisis.

Attention, dans un fichier, on ne peut pas stocker une chaîne de caractères de longueur quelconque, puisque tous les éléments du fichier doivent avoir la même taille. Vous définirez donc des chaînes de longueur fixe pour chaque champ de client.

Les différents champs de client ne seront pas des chaînes *tout court*, mais des chaînes de longueur définie (laquelle ? À vous de voir). Revoyez éventuellement le paragraphe 4C3.

✎Exercice 41✎

Écrivez un autre programme parcourant le fichier que nous venons de créer et affichant les clients dans l'ordre inverse (le dernier puis l'avant-dernier jusqu'au premier).

Modifiez ensuite l'adresse du 2^{e} client qui vient de déménager (vous mettrez une nouvelle valeur quelconque).

Affichez de nouveau le contenu du fichier, cette fois du premier au dernier élément.

Pour tous ces traitements, vous utiliserez les routines de manipulation de l'indice. Votre code doit donc être valable que le fichier contienne trois clients ou cinq cents.

Attention, ce n'est pas si simple…

4 C 12. Ajout d'un élément

Pour ajouter du texte à la fin d'un fichier texte, il fallait l'ouvrir en mode *Append*. Eh bien, avec les fichiers binaires, c'est la même chose ! *Append* vous positionne à la fin du fichier, vous pouvez donc écrire directement. Vous pouvez également l'ouvrir en mode *random* et accéder à la fin du fichier, mais c'est un peu moins simple.

✎Exercice 42✎

Écrivez un programme créant un fichier de 5 entiers, fermant le fichier puis lui ajoutant 5 autres entiers.

Travaux dirigés 2 : Fichiers !

L'objet des exercices qui suivent est de mettre en œuvre de façon systématique les concepts vus dans le cours pour que cela devienne un automatisme.

Plus que de simples exercices d'application, ils sont plutôt un complément de cours. Ne les bradez pas !

Enfin, faites les exercices dans l'ordre, la difficulté étant croissante.

Résumons notre savoir des séquences 1 à 8 :

- tout ce que nous avons vu dans les six premières séquences est acquis ;
- nous connaissons les objets importants de VB et nous devons être capables d'utiliser les autres au débotté (propriétés, méthodes et événements) ;
- les fichiers n'ont plus de secret pour nous.

Exercice 1 : Pérennité de la zone de liste

Chose promise, chose due. Reprenez le corrigé de l'exercice 30 et modifiez-le pour que :

- au démarrage de l'application, les données à afficher dans la zone de liste soient lues dans un fichier ;
- à la fermeture, le contenu de la zone de liste soit enregistré dans le même fichier.

Exercice 2 : Client pérenne

Nous allons reprendre l'exercice 6 du premier TD. Cette fois, plus question de saisir le client à chaque exécution du programme. Nous allons le saisir dans un fichier.

Voici ce que je vous demande :

- partez du programme de l'exercice 6 du TD précédent ;
- quand on clique sur *Enregistrer*, je veux que le client saisi (on suppose que tous les champs sont correctement saisis) soit ajouté dans un fichier de clients. Jouez le jeu et prévoyez tous les cas (le fichier existe déjà ou n'existe pas encore) ;
- en cliquant sur *Vider*, les différents contrôles doivent être vidés (aucun accès n'est fait au fichier) ;
- en cliquant sur *Afficher*, il faut que le premier client contenu dans le fichier (s'il existe) soit affiché dans le formulaire. Vous conserverez le calcul de l'ancienneté.

Attention, pour pouvoir stocker le client dans un fichier, il faut indiquer la longueur de la chaîne de caractères pour chaque champ (avec le caractère *).

Exercice 4 : Clients pérennes 1

Dans l'exercice précédent, je stockais plusieurs clients dans le fichier mais je ne pouvais afficher que le premier.

Nous allons doter notre application de boutons permettant de naviguer dans le fichier, donc de pouvoir afficher les différents clients stockés. En fait, les deux boutons permettront de se déplacer dans le fichier, chaque clic sur ces boutons devant en plus afficher le client sélectionné. (Le bouton *Afficher* disparaît donc.)

Pour réaliser cela, quelques ajustements vont être nécessaires. Avant toute chose, vous ajouterez deux boutons, l'un permettant d'accéder au client précédent, l'autre au suivant.

Ensuite, il va falloir modifier légèrement l'organisation de notre code.

Dans la correction de l'exercice précédent, les accès au fichier étaient cachés dans chaque sous-programme : les deux ouvraient le fichier, faisaient ce qu'ils avaient à faire puis le fermaient. Impossible donc de connaître la position courante dans le fichier et de travailler en conséquence.

Deux solutions sont possibles :

- en utilisant les événements adéquats, on peut ouvrir le fichier à l'ouverture du programme, le maintenir ouvert tout au long du programme (les différents sous-programmes pourront alors y accéder directement et l'indice courant sera connu) et le fermer en quittant le programme ;
- on conserve le fonctionnement actuel (ouverture puis fermeture à chaque accès) mais on ajoute une variable accessible tout au long du programme et indiquant l'indice de l'élément actuellement affiché.

Je vous demande de mettre en œuvre la première solution (en n'oubliant pas les boutons de déplacement). Elle n'est pas difficile puisque nous avons vu que le fichier était constamment ouvert en mode *Random*.

Exercice 4 : Clients pérennes 2

La première solution (que vous venez d'implanter) est apparemment plus simple et, c'est sûr, plus rapide car elle n'engendrera qu'une ouverture et une fermeture du fichier. Ceci dit, je ne l'aime pas beaucoup car laisser des ressources ouvertes en permanence n'est pas satisfaisant.

Mettez en œuvre la seconde solution.

Exercice 5 : Clients pérennes 1 + affichage agréable

Lorsque vous testez les deux programmes précédents, vous constatez à n'en pas douter un léger problème : lorsque l'on est sur le premier client et que l'on veut aller sur le précédent (qui n'existe pas), le programme plante. Idem si l'on veut accéder au client suivant le dernier.

Cela n'est pas très bien. En manipulant les *Seek*, *LOF* et autres *Enabled* (propriété du bouton), essayez d'empêcher cela. En clair :

- lorsque je suis sur le premier client, je ne dois pas pouvoir cliquer sur le bouton *Précédent* ;
- lorsque je suis sur le dernier client, je ne dois pas pouvoir cliquer sur *Suivant*.

Corrigez le programme de l'exercice 3 pour régler ce problème.

Exercice 6 : Clients pérennes 2 + affichage agréable

Corrigez le programme de l'exercice 4 pour régler le problème de *Précédent* et *Suivant*.

Séquence 9

Mise en forme de l’application

Nous allons travailler sur l’ergonomie des applications en apprenant à définir des barres de menus et d’outils.

Contenu

Tout ce qui s’applique aux barres de menus et d’outils.

Capacités attendues

Créer des menus.

1. Barres de menu et barres d'outils

1 A. Introduction

Pour le moment, nous ne savons proposer à l'utilisateur que des boutons pour qu'il puisse lancer des traitements.

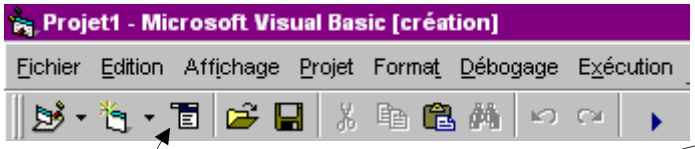
Ah, si nous pouvions ajouter à nos formulaires des barres de menus comme dans les applications professionnelles !

Eh bien, vous pouvez vous réjouir, nous allons le faire. VB vous propose un assistant pour simplifier cette tâche.

1 B. La barre de menus

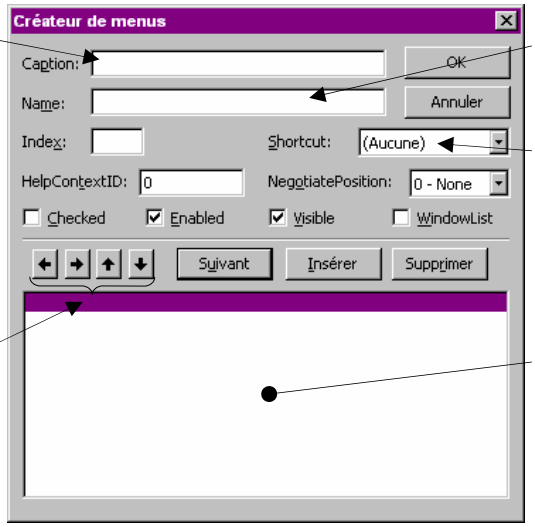
1 B 1. Comment faire

L'ergonomie Windows impose que toute commande de l'application soit accessible par la barre de menu. VB vous propose le « créateur de menus » pour la concevoir très simplement :



Cliquez ici pour lancer le créateur de menus.

Voici comment le créateur se présente :



Ici, vous mettrez la légende de votre commande. C'est elle qui s'affichera dans le menu.

Le nom de la commande sera utilisé en interne, notamment pour générer l'événement click.

Vous associerez un raccourci clavier aux commandes les plus fréquentes.

Ces boutons permettent de définir les imbrications de commandes dans les menus.

Vous visualiserez ici votre arborescence de menus.

1 B 2. Démonstration

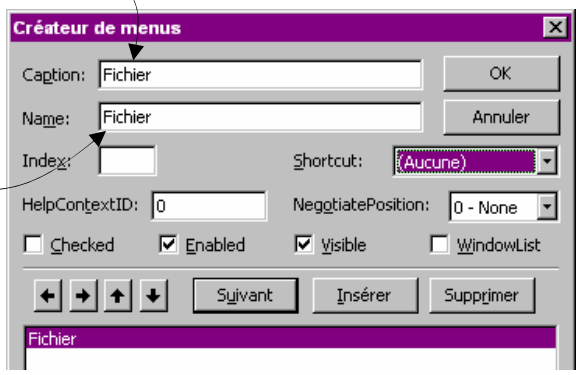
Je vais créer un menu *Fichier* contenant la commande *Quitter* et un menu *Édition* contenant *Couper*, *Copier* et *Coller* avec leur raccourci habituel.

Je vais vous présenter cette construction en trois étapes. Elles n'ont rien d'obligatoire, vous pouvez travailler dans un ordre différent du mien.

1re étape

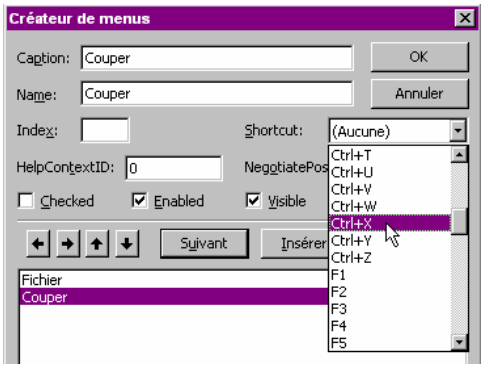
Je crée *Fichier* :

Je mets la même valeur pour la légende et le nom. C'est plus rationnel.



Je clique alors sur *Suivant* pour entrer les autres commandes. Allez, je suis un rebelle et vais toutes les rentrer successivement dans n'importe quel ordre. Je les organiserai après.

Voici l'exemple de *Couper*. Notez que je lui affecte le traditionnel raccourci *Ctrl+X* :



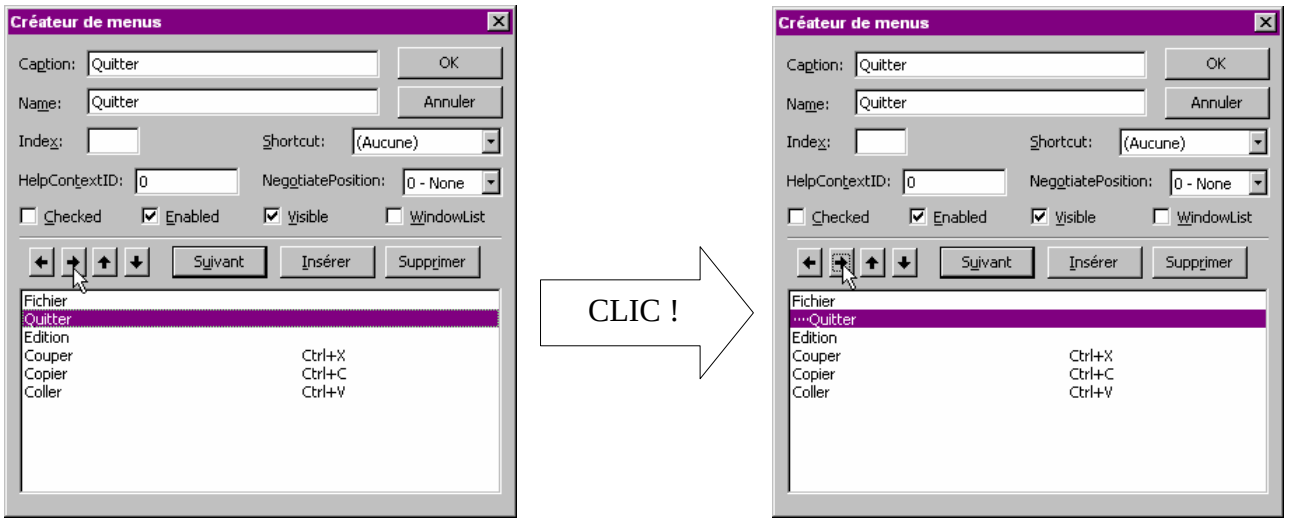
2e étape

Une fois toutes les commandes rentrées, je les réorganise pour qu'elles soient dans l'ordre (chaque menu de gauche à droite et, juste après chaque menu, ses différentes commandes).

Comment faire ? C'est très facile, je clique sur chaque commande puis je la fais monter ou descendre avec les flèches.

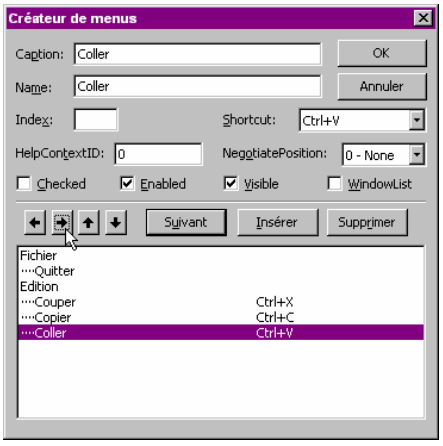
3e étape

Pour le moment, je n'ai que des menus et aucune commande. Pour dire que *Quitter* est une commande de *Fichier*, je vais sélectionner cette commande et la décaler d'une indentation vers la droite avec la flèche :



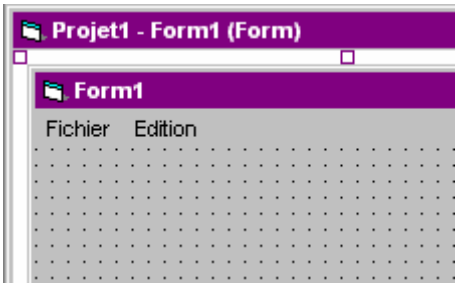
Les « … » précédant *Quitter* sont éloquents, non ? Pour avoir des sous-menus (soit des menus dans des menus), il suffirait de rajouter un niveau d'indentation.

Je m'occupe de toutes les commandes et j'obtiens :

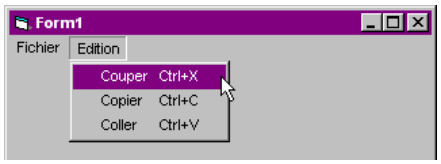


4e étape

Eh bien, j'ai terminé. Je peux cliquer sur OK et contempler mon formulaire automatiquement doté de son menu :



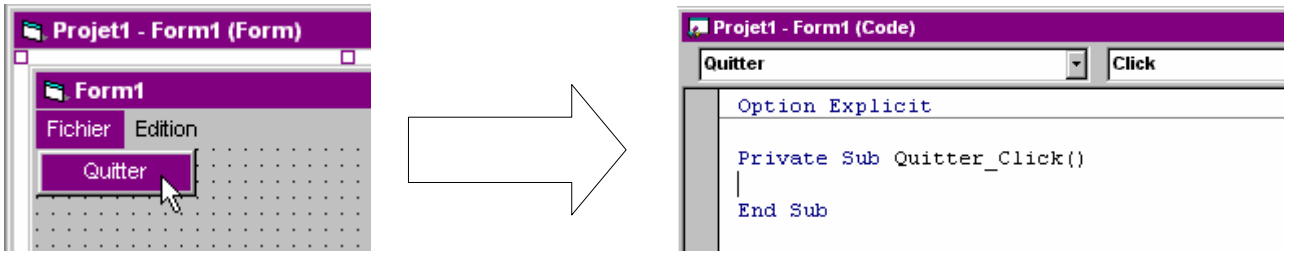
Si j'exécute l'application, je peux accéder aux sous-menus :



1 B 3. Le code

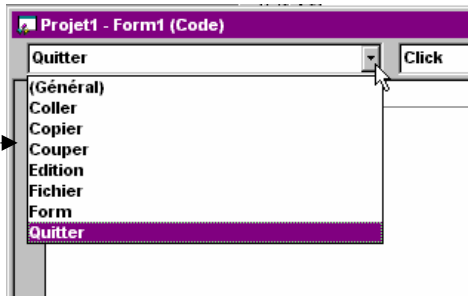
Le problème, c'est que mes menus ne font rien : je les ai créés mais aucun code ne leur est affecté.

Comment faire ? Eh bien, il suffit de sélectionner une commande en mode création du formulaire en cliquant dessus. VB génère alors l'événement *Click* associé et vous propose d'entrer le code :



Notez que vous pouvez passer par la liste modifiable de droite pour accéder directement aux différentes commandes. Chacune n'a qu'un événement, le *click* :

Si je sélectionne une de ces commandes, j'active son click.



Pour quitter une application, on utilise l'instruction *End*. Je vais m'en servir dans le code de la commande *Quitter* :

```
Option Explicit

Private Sub Quitter_Click()
   End
End Sub
```

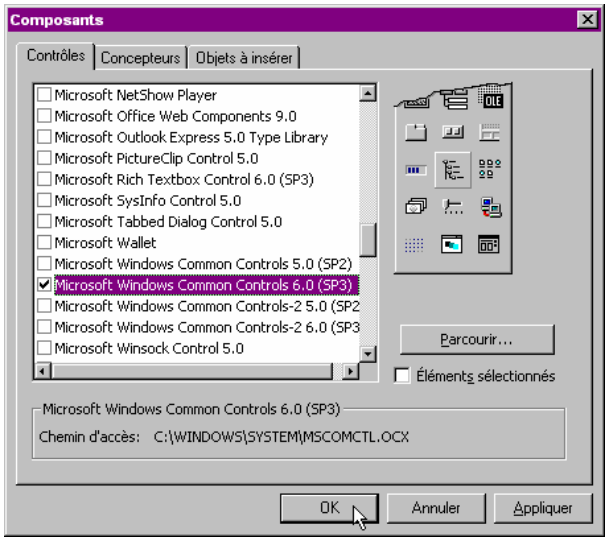
Vérifiez… cela marche !

1 C. Ajoutons une barre d'outils

Notre objectif va être de proposer une barre d'outils permettant de réaliser les opérations *Couper*, *Copier* et *Coller*. La norme Windows impose que les commandes les plus fréquentes soient dupliquées dans la barre d'outils. Elles peuvent donc être accessibles par les menus ou cette barre.

1 C 1. Le composant

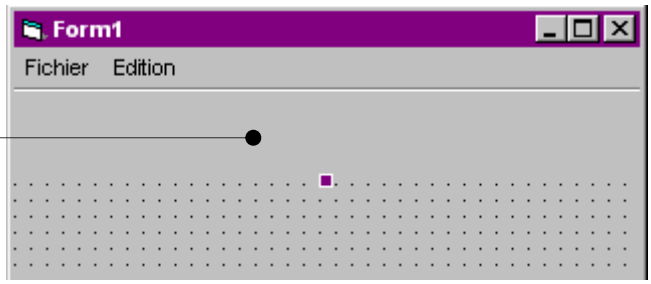
La barre d'outils n'est pas un composant standard de VB. Il faut installer le groupe de contrôles *ActiveX Microsoft Windows Common Controls* grâce à la commande *Projet/Composants…* :



Une fois la manipulation faite, vous accédez à quelques nouveaux contrôles, dont *ToolBar* (barre d'outils) dont nous avons besoin :

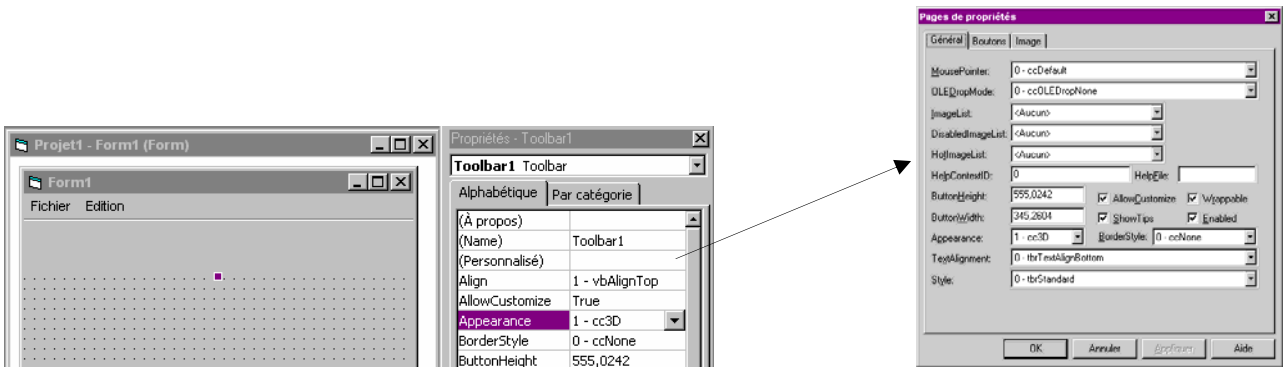
Elle est là, la petite mémère !

Prenez ce contrôle *ToolBar* et placez-le sur votre feuille. Il s'installera automatiquement là où il doit être :



La barre est là.

Si vous sélectionnez la barre d'outils, vous avez accès à ses propriétés, dont *(Personnalisé)* qui lance un assistant quand vous cliquez sur les « … » :

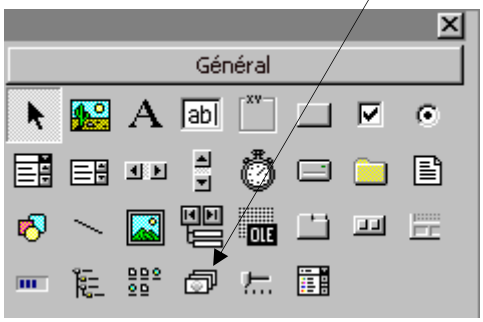


Cette boîte de dialogue est très riche, d'autant qu'elle possède trois onglets. Mais bon, elle est facile d'emploi. Enfin… *elle n'est pas trop difficile* serait plus exact.

Une barre d'outils doit proposer des commandes par l'intermédiaire de boutons contenant un icone[1]. Votre application doit donc stocker des images qui seront distribuées à la barre d'outils.

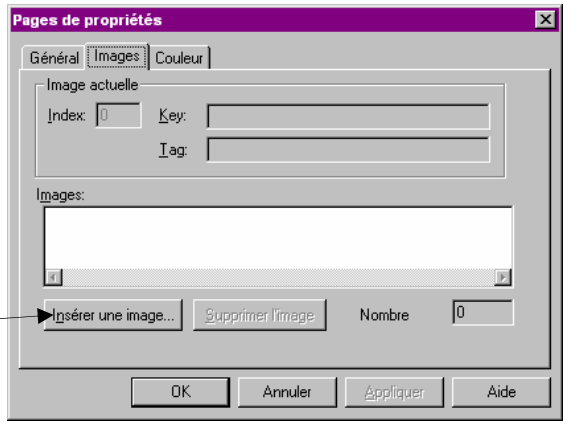
1 C 2. La liste d'icones

Pour stocker des images, rien ne vaut le composant *ImageList* (liste d'images), qui a été installé en même temps que *ToolBar* :



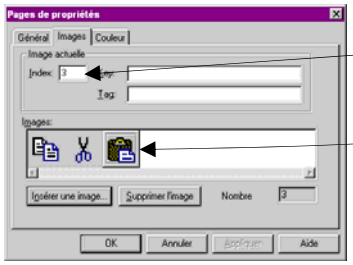
Placez un *ImageList* dans votre feuille puis accédez à sa propriété *(Personnalisé)*. C'est maintenant une habitude, vous cliquez sur « … » et obtenez une boîte de dialogue. Allez dans l'onglet *Images*. Vous obtenez :

Il faut maintenant insérer les images dans la liste. Pour cela, cliquez sur ce bouton.



Il faut ensuite récupérer des images sur le disque, d'où qu'elles viennent (d'Internet, de VB ou de dessins faits vous-même).

Ne possédant pas plus de talent graphique que de cheveux, je vais me contenter de récupérer des icones fournis par Microsoft (sur mon disque, elles se trouvent dans *C:\Program Files\Microsoft Visual Studio\Common\Graphics\Bitmaps\OffCtlBr\Small\Color*). Je récupère *Copy.bmp*, *Cut.bmp* et *Paste.bmp* :



L'icone Coller *(sélectionné sur cette copie d'écran) possède le numéro* 3.

L'icone Copier *aura le numéro* 1 *et* Couper *le numéro* 2. *Nous aurons besoin de ces numéros pour indiquer les images à mettre sur chaque bouton de la barre d'outils.*

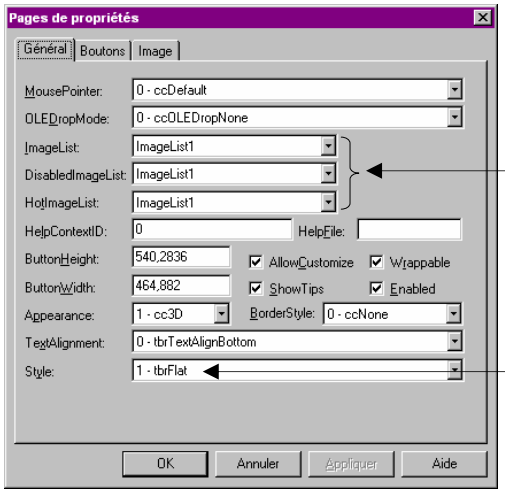
[1] À ne pas confondre avec un**e** ic**ô**ne, peinture sur bois religieuse.

Les images sont maintenant disponibles pour générer la barre d'outils. Je retourne donc à la propriété *(Personnalisé)* de cette dernière.

Voici les différentes modifications à faire. Je ne vous présente que ce qui est nécessaire. Pour aller plus loin, étudiez dans l'aide les autres rubriques paramétrables.

1 C 3. Paramétrage de la barre d'outils

1re étape, l'onglet *Général*

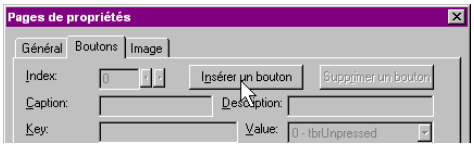


J'ai indiqué trois fois mon contrôle ImageList *pour que la barre d'outils aille chercher les images dedans. Vous pouvez mentionner trois listes d'images différentes pour que l'icone affiché change lorsqu'il est désactivé ou que vous êtes dessus. Nous restons sobres en ayant toujours le même dessin.*

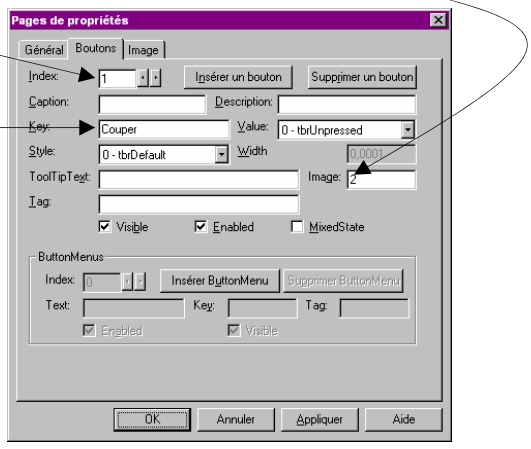
J'ai mis le style plat (flat) qui fait plus moderne.

2e étape, l'onglet *Boutons*

Je vais insérer puis paramétrer chaque bouton :



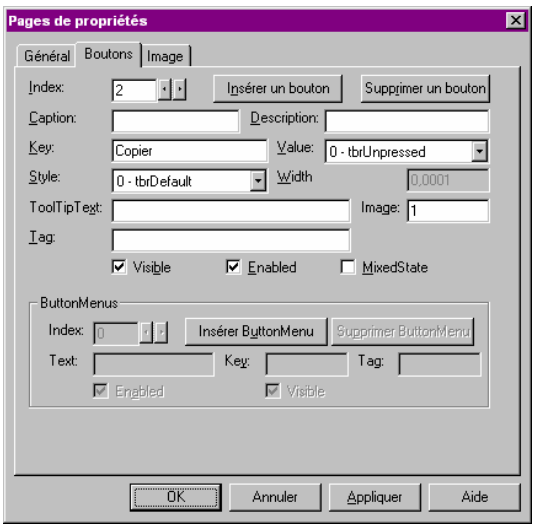
Mon premier bouton sera *Couper*. L'icone correspondant est le 2e dans ma liste d'image.

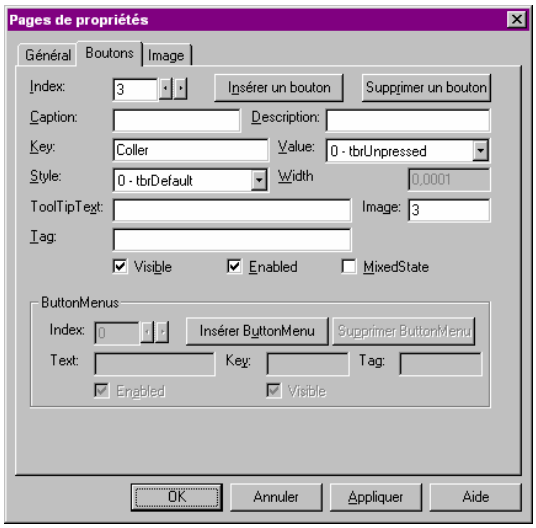


Key *permettra de distinguer facilement les différents boutons lors de l'écriture du code.*

Vous noterez que je laisse la propriété *Caption* vide. En effet, une barre d'outils n'est pas sensée contenir de texte.

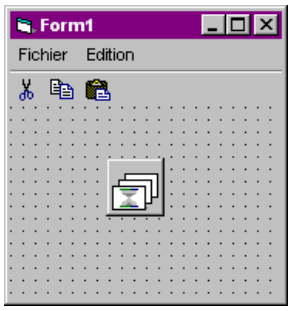
Je configure de même les deux bouton suivants (*Copier* et *Coller*) :



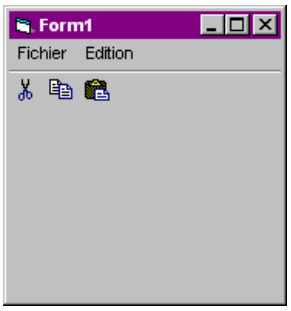


Le travail est fini. Vous pouvez fermer la fenêtre et admirer votre œuvre :

mode conception



mode exécution



Testez votre œuvre. N'y a-t-il rien qui vous gène ? Les commandes *couper*, *copier* et *coller*, que ce soit en barre de menu ou en barre d'outils, ne fonctionnent pas encore puisque aucun code n'a été écrit.

1 C 4. Écrire le code de la barre d'outils

Cela reste très rustique sous VB, comparé au raffinement extrême de la gestion des menus sous Delphi.

En effet, sous VB, vous n'avez qu'un contrôle (la barre d'outils) : les différents boutons sont transparents au niveau de l'interface. Vous devrez donc détecter l'événement *Click* sur la barre d'outils puis, avec un monstrueux *select case* sur la propriété *Key* du bouton passé en paramètre, exécuter le code correct.

Au fait, je vous ai dit précédemment que toute commande de la barre d'outils devait se trouver dans la barre de menu. Le code est donc nécessairement déjà écrit. Notre *select case* va pouvoir se contenter d'appeler ce code.

Voici tout le code de mon application :

```
01 Option Explicit
02
03 Private Sub Coller_Click()
04     MsgBox ("Code de Coller")
05 End Sub
06
```

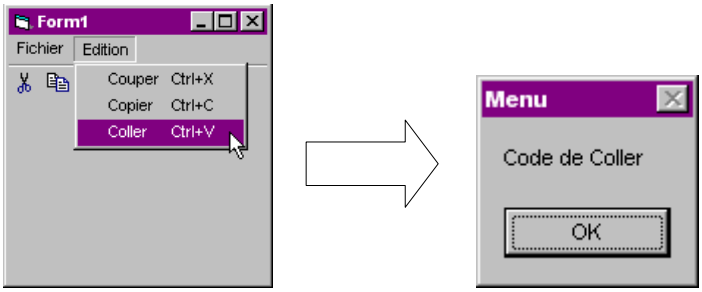
```
07 Private Sub Copier_Click()
08    MsgBox ("Code de Copier")
09 End Sub
10
11 Private Sub Couper_Click()
12    MsgBox ("Code de Couper")
13 End Sub
14
15 Private Sub Quitter_Click()
16    End
17 End Sub
18
19 Private Sub Toolbar1_ButtonClick(ByVal Button As MSComctlLib.Button)
20    Select Case Button.Key
21       Case "Couper"
22          Couper_Click
23       Case "Copier"
24          Copier_Click
25       Case "Coller"
26          Coller_Click
27    End Select
28 End Sub
```

Vous aurez remarqué que :

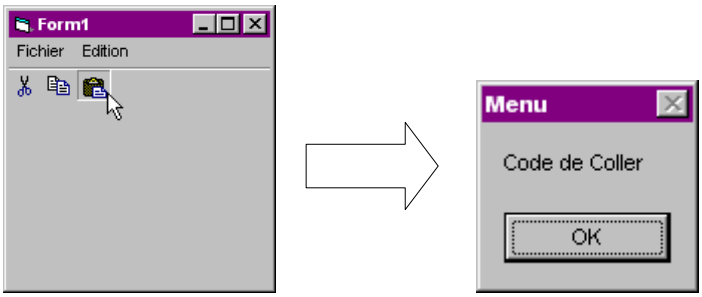
- lignes 3 à 17, il y a le code des commandes de la barre de menus ;
- lignes 19 à 28 se trouve le code du click sur la barre d'outils avec l'aiguillage selon la propriété *Key* du bouton recevant l'événement pour exécuter le code correct.

Vérifions.

Je colle depuis le menu :



Je colle depuis la barre d'outils :



Eh bien, c'est parfait !

Au fait, pourquoi me suis-je contenté de *MsgBox* et n'ai pas pris la peine d'implanter réellement *Couper* et ses amis ? Car tout dépend de ce que vous voulez couper : le code ne sera pas le même selon que vous manipulez du texte, une image…

2. Applications MDI et SDI

Lorsque vous réaliserez des applications exploitant plusieurs feuilles, vous pouvez avoir envie d'utiliser la technique MDI ou SDI.

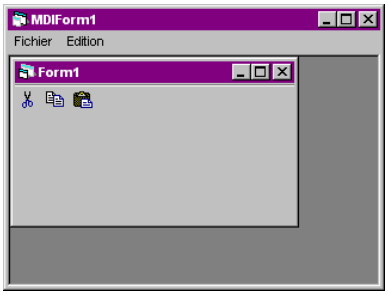
Voyons cela. En fait, vous avez deux possibilités d'applications :

- les applications SDI sont les plus simples à concevoir. Tout projet (y compris ceux que nous avons déjà réalisés) sont SDI, soit *Single Document Interface* (interface de document unique). Cela signifie que les différentes feuilles affichées sont au même niveau (donc contiennent des données sans rapport de hiérarchie). On peut donc les agencer comme l'on veut ;
- les applications *MDI* (*Multiple Document Interface*, interface de document multiple) permettent d'ouvrir plusieurs feuilles identiques (contenant le même type de données) contenues dans une feuille principale. Deux exemples classiques : Word et Excel.

Par défaut, un projet est SDI. Pour faire une application SDI, vous devez :

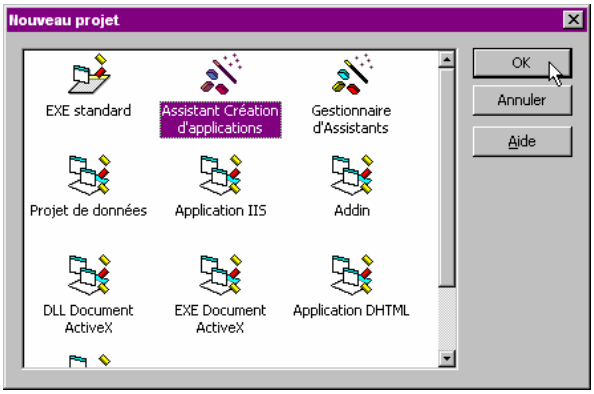
- ajouter une feuille container MDI avec la commande *Projet/Ajouter une feuille MDI*. Vous ne pouvez avoir qu'une feuille de ce type par projet, par définition du MDI. Elle devient automatiquement la feuille de démarrage ;
- les autres feuilles, dites feuilles enfant, auront leur propriété *MDIChild* à *true*.

Voici un exemple d'application MDI (j'ai repris la feuille faite avec les menus, je l'ai définie comme feuille enfant et j'ai ajouté une feuille MDI au projet, et c'est tout… cinq clicks m'ont suffi) :



Vous noterez que la norme MDI impose que la feuille MDI container contienne la barre de menus, la fenêtre fille ne gardant que sa barre d'outils (je n'ai rien fait en ce sens, c'est VB qui l'a fait automatiquement).

Pour créer une application MDI ou SDI, vous pouvez passer par la commande *Fichier/Nouveau projet*. Vous obtenez alors la boîte suivante :



Ne choisissez plus *EXE standard* mais *Assistant Création d'applications*. Il vous proposera de réaliser le canevas d'une application MDI ou SDI au choix. Faites-le puis étudiez le code !

Travaux dirigés 3 :

Application complète

Résumons notre savoir des séquences 1 à 9 :

- nous savons tout faire ;
- euh, c'est tout.

L'objet de ce TP va être d'enrichir notre programme de gestion des clients (exercice 6 du TD 2) pour avoir une application complète (menus, différents formulaires…).

Voici le cadre de travail que nous allons informatiser.

L'objet de l'application n'est pas de gérer les clients. Non. Il s'agit de gérer les appels des clients car nous sommes dans le service commercial.

L'idée est la suivante : lorsqu'un client appelle la société, ce peut être pour différents motifs : une question technique, une demande de catalogue, une plainte… Il a été décidé d'archiver tous les appels de chaque client avec la réponse qui a été donnée. Il sera alors possible d'interroger l'application pour connaître les différents appels d'un client donné, de connaître les appels restés sans réponse…

En clair, nous allons développer une application de suivi de la relation client.

Chaque exercice va nous faire progresser dans la réalisation de l'application. À l'issue de chacun d'eux, votre application doit se compiler. Si ce n'est pas le cas, c'est que vous avez des erreurs à corriger (essayez de le faire vous-même avant d'étudier la solution).

Exercice 1 : Définissons les types

Les clients seront stockés dans un fichier appelé *Fichier Client*. Les appels seront stockés dans un fichier appelé *Fichier Appel*. Vous devrez donc déclarer des types *Client* et *Appel* puis des fichiers associés.

Les clients sont toujours les mêmes que dans les exercices précédents (nom, prénom, adresse, téléphone et date de première commande). Nous rajoutons un numéro qui identifiera de façon unique le client. Ainsi, je ne parlerai pas du client *Jean-Yves Février* (car, le monde étant vaste, il peut y avoir plusieurs clients ayant ce nom), mais du client numéro 7. À chaque fois que vous saisirez un client, vous devrez mettre un numéro nouveau (nous y reviendrons).

Comment caractériser un appel ? Eh bien, un appel sera défini par :

- une date ;
- une heure ;
- le numéro du client qui l'a passé ;
- la raison de l'appel ;
- la réponse apportée.

Ce n'est pas clair ? Voici un exemple des deux fichiers :

Contenu du fichier *fichier client*					
NUMERO	NOM	PRENOM	ADR	TELEPHONE	DATE 1RE
1	Teckel	Nina	715, rue Mitterrand	03 44 55 11 22	19/12/1998
2	Février	Jean-Yves	200, rue de l'Ardoux	03 44 55 44 33	14/07/1996
3	Kaninchen	Raoul	4, rue des Arbués	02 33 44 55 44	22/11/2002
4	Lechat	Pollen	Petite rue bâclée	01 02 03 04 05	05/03/2003
…					

La seule chose importante, c'est que les numéros soient uniques. Leur valeur exacte n'a aucune importance.

Le fichier *fichier appel*				
DATE	HEURE	NUM	RAISON	REPONSE
13/05/2001	13:45	2	impossible de démarrer le PC	retour en atelier (sous garantie)
13/05/2001	16:34	2	faut-il renvoyer l'écran ou que l'unité centrale	suite à test par le client, que l'écran (le PC fonctionne, c'est l'écran qui est défectueux)
19/07/2002	08:00	9	demande extension de garantie	contrat envoyé
23/11/2003	13:46	10	Remise si achat de 40 PC réf 445.Niok ?	
03/12/2003	09:27	2	modem ne fonctionne plus	plus sous garantie, devis de prise en charge envoyé

Vous voyez que le quatrième appel enregistré dans le fichier n'a pas eu de réponse. Cela signifie que le centre d'appels ayant traité la demande n'avait pas l'information. Un responsable va étudier ce soir la liste des appels sans réponse et contactera les clients pour satisfaire à leur demande (le fichier étant alors mis à jour).

On remarque que le client 2 a appelé plusieurs fois. Au fait, qui est ce client 2 ? Pour connaître son nom et ses coordonnées, il suffit d'aller le chercher dans *fichier client*. (On parcourt le fichier jusqu'à trouver le client possédant le numéro 2[1].)

Travail à faire :

1. Dans un nouveau projet, ajoutez un module. Comme un module n'est lié à aucune feuille, il ne contiendra que du code et des déclarations. C'est là que vous définirez les fichiers et les types. L'intérêt, c'est que toute feuille du projet y aura accès directement.
2. Définissez au bon endroit les types *Client* et *Appel* puis deux descripteurs de fichier, l'un pour stocker les clients, l'autre les appels.
3. Enregistrez ce module sous le nom *Declarations*.
4. (Rappel : compilez votre programme en l'exécutant. S'il y a des erreurs, corrigez-les.)

Exercice 2 : Configuration des fichiers

Avant de parler de client, je voudrais que les deux fichiers (celui des clients et celui des appels) existent toujours au démarrage de l'application, quitte à ce qu'ils soient vides. Comment faire cela et où le faire de façon propre ?

Voyez-vous l'intérêt de faire cela ?

[1] Et là, surprise, le client 2, c'est moi ! Non, en fait, c'est un homonyme.

Exercice 3 : Gestion des appels : ergonomie

Ajoutez une nouvelle feuille permettant de gérer les appels. Dans un premier temps, nous allons uniquement nous préoccuper de l'interface. Le code sera ajouté ensuite.

La feuille doit posséder les contrôles pour saisir ou afficher un appel (horaire, numéro du client, raison et réponse).
Pour le moment, on ne saisit que le numéro du client, ce qui n'est pas très pratique. Nous verrons plus tard comment optimiser cela.

Exercice 4 : Gestion des appels : menus

À quoi va me servir ma feuille des appels ? Eh bien :

- à saisir un appel, puis à l'enregistrer dans le fichier. Je dois pouvoir annuler ma saisie si nécessaire ;
- à consulter les différents appels en navigant dans le fichier ;
- à lister, dans un fichier texte, tous les appels sans réponse (l'idée est que le responsable imprimera ce fichier pour le traiter).

Dans le TD précédent, la consultation et la saisie étaient faites par l'intermédiaire de boutons de commande. Ce n'est plus satisfaisant depuis l'excellente séquence 9.

Vous ajouterez donc une barre de menus, voire d'outils, pour réaliser ces différentes opérations. Ne faites pas le code, ce sera pour l'exercice 6.

Exercice 5 : Définissons les conditions initiales

Lorsque l'application est lancée, la feuille de gestion des appels doit se lancer et afficher le **dernier** appel stocké dans le fichier car c'est le plus récent.

Bien entendu, si le fichier est vide, rien n'est affiché.

Exercice 6 : Gestion des appels : code

Écrivez le code des actions définies dans l'exercice 4. Ne vous occupez pas de la désactivation des commandes traduisant par exemple que l'on ne peut pas accéder à l'appel suivant le dernier. Ce sera pour l'exercice suivant.

Voici le descriptif de ce que je souhaite :

- lorsque l'on choisit la commande *Nouveau*, la feuille doit devenir vierge. L'utilisateur peut alors saisir un appel. Il devra ensuite cliquer sur *Valider* (et l'appel sera enregistré) ou choisir *Annuler*, auquel cas sa saisie est ignorée et il retrouve l'appel sur lequel il était ;
- pour *Précédent* et *Suivant*… cela doit être clair.

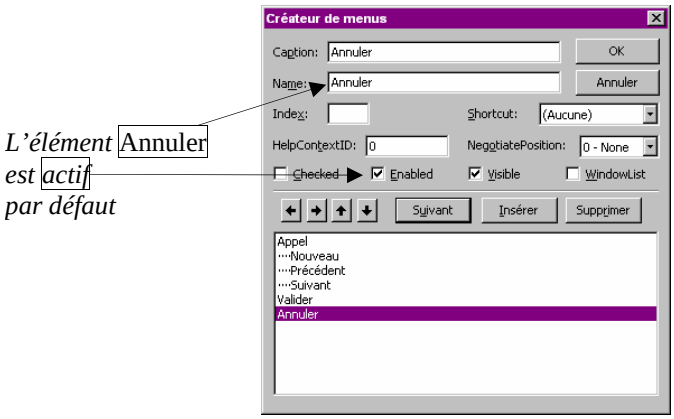
Ah, j'oubliais… lorsque je veux saisir un nouvel appel, j'aimerais bien que les date et heure systèmes initialisent automatiquement le contrôle *Horaire*.

Exercice 7 : Gestion des appels : ergonomie

Il est temps de mettre de l'ordre dans l'activation des différents éléments du menu. Je veux que l'on ne puisse lancer une action que si cela a un sens :

- on ne peut pas faire *Précédent* si l'on est sur le premier appel (ou s'il n'y en a pas) ;
- on ne peut pas faire *Suivant* si l'on est sur le dernier appel (ou s'il n'y en a pas) ;
- si l'on est en train de saisir un nouvel appel, on ne peut faire que *Annuler* ou *Valider* ; réciproquement, ces commandes ne sont pas accessibles dans d'autres situations.

Comment faire cela ? C'est aussi simple que lorsque nous l'avons fait avec les boutons dans la séquence 9 : chaque élément de menu possède une propriété *Enabled* qui vaut *true* ou *false*.



Exercice 8 : Gestion des appels : barre d'outils

Ajoutez une barre d'outils ne contenant que des icones et gérez leur désactivation comme pour le menu.

Exercice 9 : Saisie d'un nouveau client

Ajoutez une nouvelle feuille permettant juste de saisir un client. Nous retrouverons donc les commandes *Valider* et *Annuler* mais ni *Précédent* ni *Suivant* ni *Nouveau*.

Cette feuille devra s'ouvrir lorsque l'utilisateur double-cliquera sur le contrôle *Client* de *GestionAppel*.

Attention !

Nous avons vu dans le sujet que le numéro de client servait uniquement à identifier de façon unique un client, ce qui permet de faire le lien entre un appel et le client associé.

Je souhaite donc que l'utilisateur ne puisse pas rentrer de numéro de client. C'est à l'application de le déterminer automatiquement. Comment ? C'est simple : comme nous ne supprimons pas de client, vous pouvez prendre comme numéro identifiant le nombre d'éléments du fichier une fois qu'il aura été ajouté.

Les deux commandes *Valider* et *Annuler* doivent fermer le formulaire. *Valider* enregistrera le client dans le fichier et initialisera automatiquement le contrôle *Client* du formulaire *GestionAppel* par le numéro du client saisi.

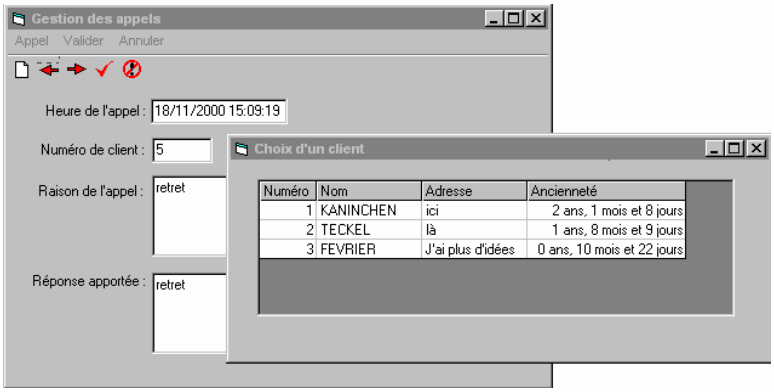
Exercice 10 : Choix d'un client existant

Je voudrais que lorsque le focus arrive sur le contrôle *Client* de *GestionAppel*, une nouvelle feuille contenant une grille (contrôle *MSFlexGrid* qu'il faut ajouter… vous le trouverez dans *Microsoft FexGrid Control 6.0*).

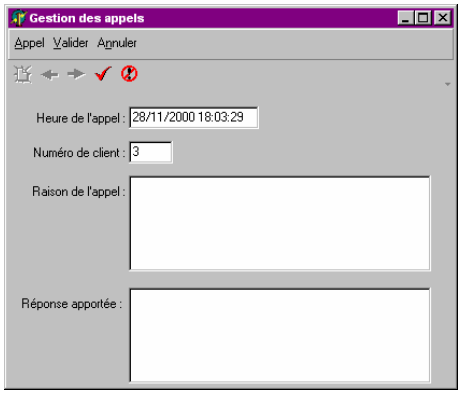
La grille devra contenir les différents clients (leur numéro, nom, adresse et ancienneté). Lorsque l'utilisateur clique sur une ligne de la grille, la fenêtre se referme et le numéro de client correspondant s'affiche dans *Client*. Vous utiliserez avec profit la propriété *TextMatrix* de la grille.

Démonstration :

J'ai voulu ajouter un nouvel appel. Comme l'horaire est saisi automatiquement, le focus passe à *Client*, ce qui déclenche l'affichage de la grille :



Si je clique sur la ligne 3 pour sélectionner mon homonyme, la grille se referme et le numéro est automatiquement affiché. Le focus passe à la zone de texte *Raison* :



Exercice 11 : Appels sans réponse

Ce n'est pas le plus difficile… Ajoutez une commande *Rapport* dans la barre de menus pour créer un fichier texte *Rapport.td3* contenant la liste des appels sans réponse.

Chaque appel tiendra sur deux lignes dans le fichier :

- première ligne, vous mettrez l'horaire de l'appel, le nom, le prénom et le téléphone du client ;
- seconde ligne, le motif de l'appel (avec une petite indentation pour la lisibilité).

Il vous faudra parcourir le fichier des appels et, pour chaque appel sans réponse, aller chercher le client correspondant. Voici un exemple de fichier *Rapport.td3* :

```
28/11/2003 18:50:06 FEVRIER Jean-Yves 33 33 33 33 33
     PC marche plus
28/11/2003 18:50:44 KANINCHEN Raoul 22 22 22 22 22
     Devis reçu non conforme
28/11/2003 18:51:39 FEVRIER Jean-Yves 33 33 33 33 33
     Aimez-vous les teckels ?
```