VISUAL BASIC

Visual Basic .NET

PARTIE	PAGES
Préambules nécessaires (et néanmoins très intéressants)	**3-4**
VB.NET, un langage objet	5-12
L'interface de Visual Studio	13-18
Premiers contrôles	19-23
Les bases du langage	24-35
D'autres contrôles	36-39
Les collections	40-47
Encore des contrôles	48-55
Événements insolites	56-63
La chasse aux bugs	64-72
Toujours des contrôles	73-76
Les graphismes	77-78
Les menus	79-80
Distribuer une application	81-83
Les fichiers texte	84-87
VB.NET et Windows	88-96
Plus loin avec les objets	97
Les applications MDI	98

Préambules nécessaires (et néanmoins très intéressants)

Avant toute autre chose, ne faites pas comme moi, ne passez pas d'entrée pour une tanche.

Donc, apprenez à prononcer correctement. Ne dites jamais : "*Vébé-poin-nette*", sous peine d'être immédiatement regardé comme le dernier des pécores attardés. Évitez "*Vébé nette*", trop commun, et qui ne vous distinguera pas suffisamment du vulgaire. Si vous voulez vraiment épater la galerie, et faire figure d'authentique informaticien, optez sans vergogne pour "*Vébé-dotte-nette*". Vous serez crédités dans l'instant

de l'aura qui entoure les choisis, les élus, les membres du cénacle ultime, bref, ceux qui détiennent un savoir ésotérique impénétrable à la majorité des mortels.

Ensuite, et pour parler de ce qui vous attend dans les pages qui viennent, ce site a été réalisé avant tout pour servir de support au cours que j'ai le plaisir, l'honneur et l'avantage de dispenser devant mes étudiants. Cela explique en bonne partie (mais n'excuse pas) ses faiblesses et ses insuffisances. Donc, plutôt que de parler de tout ce qu'il contient, commençons par préciser d'avance tout ce qui y manque :

Ce n'est pas un manuel pour débutants en programmation. Le site s'adresse à un public ayant maîtrisé, au moins un minimum, les rudiments de l'algorithmique et de la programmation procédurale. Au cours des explications qui vont suivre, ces rudiments sont supposés connus. Si ce n'est pas ton cas, petit scarabée, passe ton chemin ! (ou plutôt, prends le temps de flâner du côté d'un sensationnel cours d'algorithmique dont on ne dira jamais assez de bien). Après quelques dizaines d'heures de torture mentale, tu auras gagné plusieurs points de karma informatique et tu pourras revenir ici sans que les choses tournent en eau de Bouddha.

Ce n'est pas un véritable cours en ligne, même si cela s'en rapproche dangereusement. Il se peut très bien que pour une autoformation, les explications soient par moments un peu courtes, voire franchement insuffisantes. Car encore une fois, ce site est avant tout un support pour des démonstrations et des exercices effectués en direct live (et sans filet).

Ce n'est pas une référence complète du langage. D'abord, parce que celle-ci existe déjà dans l'aide du logiciel, ainsi que dans maints sites spécialisés. Ensuite, parce que mon propos d'enseignant n'est pas de faire ingurgiter quelques centaines d'instructions, de fonctions ou de détails techniques pointus, mais de faire comprendre les mécanismes fondamentaux d'un langage orienté objet et interface graphique. Je me suis donc limité aux aspects que j'ai estimés essentiels, et un programmeur VB devra donc nécessairement aller fouiner ailleurs pour trouver tous les détails techniques qui pourront lui être utiles.

Les problèmes liés au traitement des bases de données sont ignorés : ceci parce que la spécialité de Master M2 dans laquelle ce cours est dispensé consacre déjà suffisamment d'autres enseignements à cet aspect.

Ce bref instant d'autocritique passé, je peux à présent, et sans fausse modestie, énumérer les principales qualités de ce merveilleux site (il en a sans doute bien d'autres, mais je compte sur mes estimés lecteurs pour les remarquer par eux-mêmes).

Il se veut pédagogique. J'ai essayé, dans toute la mesure du possible, de m'exprimer dans le langage le plus abordable qui soit. Les termes techniques seront expliqués au fur et à mesure, et seront souvent illustrés par de fulgurantes métaphores aussi poétiques que suggestives. Ou alors, à la rigueur, par de petits dessins très jolis et toujours de bon aloi.

Visual Basic .NET

Il est en français, et qui plus est sans - trop - de fautes d'orthographe, ce qui, vous en conviendrez sans doute, ne gâte rien, surtout quand on est francophone.

Il présente un certain nombre d'exemples, sous forme d'exercices corrigés.

Il contient nombre de saillies drolatiques d'une telle finesse que son auteur a dû à plusieurs reprises décliner des offres de participation aux Grosses Têtes.

Last but not least, **il est entièrement gratuit** (ce à quoi les mauvais esprits rétorqueront qu'on en a effectivement pour son argent).

Remarque essentielle :

Ce dernier point ne doit nullement vous empêcher, si le coeur vous en dit, de témoigner votre gratitude en m'invitant au restaurant.
Voire carrément chez vous, surtout si vous habitez dans un endroit ensoleillé, près d'un océan où batifolent des poissons tropicaux.
J'encourage comme il se doit ces manifestations de gratitude bien compréhensibles, et je me ferai un point d'honneur à répondre favorablement à toute proposition de ce genre.
Et pour me contacter, c'est très facile.

Bon, eh bien sur ce, bonne lecture, et bon apprentissage à tous, en gardant à l'esprit la profonde maxime du regretté professeur Shadoko : "*La plus grave maladie du cerveau, c'est de réfléchir*".

Partie 1

VB.NET, un langage objet

Visual Basic .NET

1. C'est quoi, un langage objet ?

Vous le savez sans doute déjà : VB.NET est un langage objet. Ce n'est pas le seul : C# (prononcez "*écharpe*"), C++ ou Java sont eux aussi des langages objets. Ce type de langage, apparu dans les années 1990, s'oppose en particulier aux langages de la génération précédente, qui étaient dits "procéduraux" (comme le Cobol, le Fortran, le Pascal, le C et bien d'autres, on ne va pas tous les citer, on n'a pas que cela à faire non plus).

La première chose à dire des langages objet, c'est qu'ils permettent de faire tout, absolument tout, ce qu'on pouvait faire avec un langage procédural.

Donc, et c'est la bonne nouvelle du jour, rien de ce que vous avez appris n'est perdu ! Les langages objet, comme leurs ancêtres les langages procéduraux, manipulent des variables, des tableaux et des structures, et organisent des instructions d'affectation, d'entrée-sortie, de tests ou de boucles dans des procédures et des fonctions. Donc, à la limite, on peut (presque) programmer en langage objet comme on programmait en langage procédural. Mais évidemment, dans ce cas, on passe à côté de ces petits riens que les langages objet possèdent en plus, petits riens qui vont nous changer la vie...

Remarque sournoise :

S'il y a quelque chose que vous ne comprenez pas dans le paragraphe qui précède, c'est vraisemblablement que vous avez doublement brûlé les étapes.
D'une part, en voulant apprendre VB.NET sans avoir appris l'algorithmique.
D'autre part, en n'ayant pas lu assez attentivement les Préambules nécessaires (ce qui n'est pas bien) ou en les ayant lu, et en choisissant de les ignorer (ce qui est encore pire).
Dans les deux cas, vous n'échapperez pas à un retour par la case départ, et vous ne touchez pas 20 000 F.

Reprenons. Les langages objet, tout en intégrant l'ensemble des capacités des langages procéduraux, possèdent deux aptitudes supplémentaires, aptitudes liées mais distinctes. Ces langages peuvent en effet :

- gérer, en plus des variables, des choses appelées **objets**.
- exploiter, via certains de ces objets, l'**interface graphique** de Windows.

On verra plus tard que par contamination, les langages objet ont tendance à tout assimiler à des objets, même des choses qui n'en sont pas spécialement, comme les bonnes vieilles variables de notre enfance. Mais n'anticipons pas, et examinons tout d'abord les deux points ci-dessus plus en détail.

1.1 Qu'est-ce qu'un objet ? Parlons comme les gens normaux

Dans la vie de tous les jours, nous sommes entourés d'objets. Certains très simples, comme un peigne, un bouton de culotte ou une matraque de CRS. D'autres très complexes comme une automobile ou une navette spatiale. Pour faire l'analogie avec les objets des langages objet, mieux vaut penser à un objet un peu compliqué qu'à un objet trop simple. Et pour cette analogie, rien ne nous oblige à penser à un objet

inanimé. Des plantes, des animaux ou des êtres humains peuvent tout à fait être assimilés aux objets que manipulent les langages.

Dans les objets, il y a deux sortes de choses qui intéressent les informaticiens.

- **ils ont des caractéristiques** (une couleur, une taille, un poids, un compte en banque, un nombre de pattes, bref, que sais-je encore). Un objet très simple peut avoir une seule caractéristique (voir aucune, mais là, on peut raisonnablement se demander s'il existe - je laisse cette question aux philosophes désoeuvrés). Un objet complexe peut avoir des centaines, voire des milliers de caractéristiques.
- **ils peuvent accomplir certaines tâches**, ou subir certaines modifications : pour une automobile, on peut ouvrir une porte, démarrer le moteur, freiner, accélérer, etc. Pour un chat, on peut le caresser, le faire marcher, le faire courir, le faire miauler, le faire cuire, etc. Pour un employé, on peut le changer de poste, de qualification, l'augmenter (c'est rare), le licencier (c'est fréquent), etc.

Tous les objets possédant les mêmes caractéristiques et avec lesquels on peut faire les même choses forment un groupe qu'on appelle en informatique **une classe**. Par exemple, tous les chiens ont une couleur de pelage, une taille, un poids, etc. Ils peuvent tous aboyer, courir, sauter, etc. Même si chacune de ces aptitudes est différente d'un chien à l'autre, et même si le chihuahua de ma cousine est assez différent du rottweiler de mon voisin, ces deux bestioles ont en commun de posséder certaines caractéristiques et de pouvoir effectuer certaines actions. Ils font donc partie d'une classe unique que j'appelle "les chiens", même si d'un chien à l'autre, et ma cousine en sait quelque chose lorsqu'elle croise mon voisin, la taille et le poids (entre autres) ne sont pas les mêmes.

En prenant le problème à l'envers : si je définis une classe par l'ensemble de ses caractéristiques et de ses actions, je peux, à partir de cette classe, fabriquer tout plein de toutous différents les uns des autres, mais qui auront en commun d'être tous des chiens (donc, répétons-le, des êtres qui partageront les mêmes caractéristiques et qui seront capables des mêmes actions).

Eh bien, les langages objet, ce sont des langages qui permettent, en plus des traditionnels variables, tableaux et structures, de gérer et de manipuler des objets (donc des classes). C'est-à-dire que ces langages permettent notamment :

de **créer des classes**, en définissant leurs caractéristiques et leurs actions

de **fabriquer des objets** à partir de ces classes

de **manipuler les caractéristiques individuelles de chacun de ces objets**, et de **leur faire accomplir les actions** dont leur classe est capable.

Tout ce que nous avons vu là peut être traduit dans des termes techniques propres aux informaticiens.

1.2 Qu'est-ce qu'un objet ? Parlons comme les informaticiens

Visual Basic .NET

Nous savons que tout programme informatique a pour but de manipuler des informations. Et nous savons que ces informations relèvent toujours, au bout du compte, de l'un des trois grands **types** simples : numériques, caractères (alphanumériques), ou booléennes.

Lorsque nous disons qu'un objet (ou une classe) possède des caractéristiques comme la taille, le poids, la couleur, etc., nous disons en fait qu'un objet (ou une classe) regroupe un certain nombre de **variables** : une variable numérique pour stocker le poids, une autre variable pour stocker la taille, une variable caractère pour stocker la couleur, etc.

Donc, première chose, un objet regroupe un ensemble de variables qui peuvent être de différents types. Et dans ce cas, on ne parlera pas des "variables" d'un objet, ni des ses "caractéristiques", mais de ses **propriétés**.

Définition :
Une propriété d'un objet est une des variables (typée) associées à cet objet.

Théorème :
Par conséquent, toute instruction légitime avec une variable est légitime avec une propriété d'un objet. Et lycée de Versailles, tout ce qui n'avait pas de sens avec une variable n'a pas non plus de sens avec une propriété d'un objet.

Jusqu'à maintenant, allez vous dire, nous n'avons fait que faire de la mousse avec du vocabulaire nouveau autour d'une notion que nous connaissons déjà comme notre poche. Parce qu'enfin, quoi, un bidule constitué de différentes variables de différents types, on n'a pas attendu les objets pour en avoir. Tous les langages procéduraux connaissent cela, mis à part qu'on ne parle pas de classe mais de **structure**, qu'on ne parle pas d'objets, mais de **variables structurées**, et qu'on ne parle pas de propriétés, mais de **champs**. Mais pour le reste, c'est tutti quanti et du pareil au même.

Les esprits chagrins qui feraient cette remarque auraient parfaitement raison... si les objets n'étaient effectivement qu'un ensemble de caractéristiques (de propriétés). Mais les objets, comme nous l'avons vu, ne sont pas que cela : ils incluent également, comme on l'a vu, des **actions** possibles. Autrement dit, un objet (une classe) est un assemblage d'informations (les propriétés) mais aussi d'instructions (les actions, que l'on appelle les **méthodes**). C'est en particulier cette présence des méthodes qui différencie une classe d'une simple structure, et un objet d'une simple variable structurée.

Définition :
Une méthode est un ensemble d'instructions (de lignes de codes) associées à un objet.

Théorème :
Un objet est un assemblage d'informations et d'instructions.
Autrement dit, c'est une variable structurée plus des méthodes.

2. La syntaxe objet

2.1 Généralités

Lorsque nous fabriquions (et nous en fabriquerons encore) une variable, nous utilisions un moule à variable, qui s'appelait un type. Pour créer une nouvelle variable, il fallait lui donner un nom, par lequel on pourrait la désigner tout au long du programme, et préciser le type utilisé pour la fabrication de cette variable. D'où la syntaxe du genre :

```
Dim Toto as Integer
```

Eh bien, en ce qui concerne les objets, le principe est exactement le même, hormis qu'on ne les fabrique pas avec un type, mais avec une classe. Si nous disposons par exemple d'une classe **Chien**, nous allons pouvoir créer autant de chiens que nous voulons. Voilà par exemple un nouveau compagnon :

```
Dim Rex as New Chien
```

On remarque la présence du mot clé **New**, propre aux déclarations d'objets, et qui différencie celles-ci des déclarations de variables. Dans le jargon du langage objet, cette instruction **New** s'appelle un **constructeur**.

Vocabulaire savant :

Créer un objet d'après une classe s'appelle **instancier** la classe.
Un objet peut aussi être appelé une **instance**, ou une **occurrence**, de la classe.

Notre classe **Chien** a forcément été créée avec un certain nombre de propriétés et de méthodes. Nous pourrons accéder à ces propriétés et à ces méthodes par la syntaxe suivante, universelle :

```
NomdObjet.Propriété
```

Et également :

```
NomdObjet.Méthode
```

Remarque fondamentale :

Dans un langage objet, on ne peut trouver que les deux syntaxes ci-dessus.
Il est absolument impossible de désigner un objet sans le faire suivre d'une propriété ou d'une méthode. C'est une faute de syntaxe.
De même, une propriété seule ou une méthode seule constituent également des fautes de syntaxe.

2.2 Propriétés

Admettons pour les besoins de la cause que notre classe **Chien** possède entre autres propriétés :

Taille (numérique)

Poids (numérique)

Couleur (caractère)

Vacciné (booléen)

Alors, je le rappelle, les règles d'utilisation de ces propriétés sont très exactement les mêmes que celles des variables. Sachant que le signe d'affectation, en Visual Basic, est le signe d'égalité, je peux donc écrire :

```
Rex.Poids = 14
```

Et je viens de fixer le poids de mon toutou à 14 kg. Pour le faire grossir de 5 kg, rien de plus simple :

```
Rex.Poids = Rex.Poids + 5
```

...et le tour est joué. Si je veux torturer la pauvre bête, je peux aussi la rendre aussi lourde que haute, en écrivant :

```
Rex.Poids = Rex.Taille
```

L'utilisation de propriétés non numériques ne pose pas davantage de problèmes :

```
Rex.Couleur = "Beige"
Rex.Vacciné = Vrai
```

2.3 Méthodes

Je le rappelle, si les propriétés sont en réalité des variables, les méthodes sont quant à elles des procédures. Lorsqu'on va utiliser une méthode, tout dépend donc de la manière dont la procédure a été écrite. En fait, tout dépend du nombre et du type des paramètres dont cette procédure a besoin pour s'exécuter. Ainsi, certaines méthodes peuvent-elles être utilisées sans passer de paramètres. D'autres en exigeront un, d'autres deux, etc.

Dans tous les cas, cependant, une méthode est un appel de procédure. C'est donc une instruction à part entière. **Nous en déduisons donc qu'il serait parfaitement absurde d'affecter une méthode, comme on affecte une propriété**. On pourra donc avoir des lignes du genre :

```
Rex.Aboyer
```

ou

```
Rex.Courir(15)
```

ou

```
Rex.DonnerlaPapatte("gauche")
```

etc.

Si vous avez compris cela, vous avez compris 45 % de ce cours. Très exactement, au millipoil près.

3. La programmation événementielle

Nous ne sommes pas au bout de peines (ni au bout de nos joies). Toute cette sombre histoire d'objets cache une autre drôle de plaisanterie, qui va radicalement transformer la manière dont les procédures d'un programme vont être exécutées, et du coup, la manière dont les programmeurs vont devoir concevoir leurs applications.

C'est le deuxième aspect dont je parlais au début de ce chapitre, aspect qui est lié à la programmation objet, mais qui ne se confond pas tout à fait avec elle. De quoi s'agit-il ?

Rappelons-nous comment notre code était organisé dans un langage procédural traditionnel. Plutôt qu'une immense procédure fait-tout, comportant des redites de code, nos applications, dès lors qu'elles devenaient un peu joufflues, étaient organisées en modules séparés : sous-procédures et fonctions (lesquelles, je le rappelle, ne sont jamais qu'un cas particulier de sous-procédures).

Parmi ces diverses procédures, il s'en trouvait une qui jouait un rôle particulier : la procédure principale (appelée **Main** dans la plupart des langages). C'est elle qui était exécutée lors du lancement de l'application. Et c'est elle qui tout au long du programme, telle un chef d'orchestre, déclenchait les autres procédures et les autres fonctions.

Le point important dans cette affaire, c'est qu'une fois l'application lancée, **une procédure donnée ne pouvait s'exécuter que si une ligne de code, quelque part, en commandait le déclenchement**. L'utilisateur n'avait aucune espèce de moyen d'influencer directement l'ordre dans lequel les procédures allaient être exécutées.

Tout ceci va être remis en question avec les langages objet. Non qu'on ne puisse plus créer encore des procédures et des fonctions, et appeler les unes via les autres par des instructions adéquates ; je le rappelle, il n'y a rien qu'on pouvait faire avec un langage procédural qu'on ne puisse faire avec un langage objet. Mais avec un langage objet, il y a des choses qu'on peut faire qui sont véritablement inédites. En l'occurrence, la grande nouveauté, c'est qu'**on va pouvoir organiser le déclenchement automatique de certaines procédures en réponse à certaines actions de l'utilisateur sur certains objets**.

Pour comprendre cela, le plus simple est de penser que certains objets ont une caractéristique particulière : ils se voient à l'écran. Ainsi, un logiciel comme Windows est en fait truffé de ce genre d'objets ! Les boutons, les menus, les fenêtres, les cases à cocher, tout cela constitue une armada d'objets visibles, et surtout, capables de réagir à diverses sollicitations de l'utilisateur via le clavier ou la souris. Les programmeurs qui ont écrit Windows ont donc prévu, et écrit, des myriades de morceaux de code (des procédures) qui se déclenchent à chaque fois que l'utilisateur accomplit telle action sur tel type d'objet. Donc, je le répète, concevoir une application événementielle, c'est **concevoir des procédures qui se déclencheront automatiquement à chaque fois que l'utilisateur effectuera telle action sur tel objet qu'on aura mis à sa disposition.**

Remarque profonde :

Dans les phrases qui précèdent, les mots "*à chaque fois que*" sont essentiels. Il faut impérativement les avoir en tête lorsqu'on écrit une application objet.
Faute de quoi on mélange tout, et en particulier, on se met à écrire des boucles là où il n'y a qu'une simple procédure.

Définition :

Une action sur un objet capable de déclencher une procédure s'appelle un **événement**.

Voilà pourquoi ce type de programmation porte le nom de **programmation événementielle**. Par ailleurs, il est à noter que les événements qu'un objet donné est capable de gérer ont été définis dans la classe qui a servi à créer l'objet.

Remarque importante :
Tous les objets capables de gérer un événement ne sont pas forcément des objets visibles à l'écran. Dans le même ordre d'idées, tous les événements ne correspondent pas forcément à des actions de l'utilisateur via le clavier ou la souris. Mais pour commencer, on peut très bien s'accommoder de ces approximations.

3.1 Le diagramme TOE

Lors de la conception d'un programme objet, une des premières tâches du développeur va donc être de concevoir l'interface utilisateur. C'est-à-dire de prévoir quels objets vont être mis à la disposition de l'utilisateur, de prévoir les actions que l'utilisateur pourra effectuer sur ces objets (les événements) et de prévoir les tâches que le programme devra accomplir lors de ces événements.

Remarque incidente :
Pour le moment, nous nous contenterons de considérer que nos applications emploient l'arsenal des objets utilisés par Windows, et uniquement cela. Ces objets (plus exactement, ces classes) seront donc tout prêts à l'emploi. Leurs propriétés, leurs méthodes et les événements qu'ils sont capables de gérer ont été définis par Windows, c'est-à-dire par Visual Basic (qui nous sert en quelque sorte d'intermédiaire vis-à-vis de Windows, qui parle un langage trop abscons).
En réalité, dans un programme Visual Basic, on peut utiliser des tas d'autres objets (classes), on peut même en fabriquer soi-même. Nous verrons cela... plus tard. Pour le moment, contentons-nous de la trousse à outils de base que Visual Basic met à notre disposition, et ce sera déjà bien suffisant.

Ainsi, nous pouvons prévoir qu'il faudra que la fenêtre de notre application se présente comme ceci ou comme cela, qu'il devra y avoir ici un bouton, là des boutons radios, ici une zone de liste, etc. Et il faut également prévoir qu'un clic sur tel bouton déclenchera tel ou tel calcul, qu'un choix dans telle liste devra mettre à jour telle ou telle zone, etc.

Le document synthétique, sous forme de tableau, qui reprend toutes ces informations, s'appelle un **diagramme T.O.E.**, pour Tâche, Objet, Événement. C'est un document préalable indispensable à la programmation d'une application événementielle. Il comporte trois colonnes. Dans la première, on dresse la liste exhaustive des tâches que doit accomplir l'application. Dans la seconde, en regard de chaque tâche à accomplir, on porte, le cas échéant, le nom de l'objet qui en sera chargé. Et dans la troisième, en regard de chaque objet, on écrit le cas échéant, mais oui, l'événement correspondant. Vous l'aviez deviné, décidément vous êtes trop forts.

Remarque importante :
Rien, absolument rien, ne dit qu'un objet doit être associé à un événement et à un seul. Il peut très bien y avoir des objets qui ne gèrent aucun événement. De même, certains objets peuvent très bien être amenés à gérer plusieurs événements différents, qui correspondront donc à autant de procédures différentes.

3.2 La syntaxe

Comment effectue-t-on la liaison entre une procédure et un événement ? C'est extrêmement simple : cette liaison figure en toutes lettres dans l'en-tête de la procédure, via le mot-clé **Handles** (que l'on peut traduire par "gère"). Ainsi, une procédure non événementielle, toute bête, toute simple, que j'appelle **Calcul**, aura la tête suivante :

```
Private Sub Calcul()
    instructions
End Sub
```

Pour que cette procédure s'exécute à chaque fois que l'on clique sur le bouton nommé **Résultat**, il suffira que la procédure se présente ainsi :

```
Private Sub Calcul() Handles Résultat.Click
    instructions
End Sub
```

Si d'aventure, plusieurs événements différents doivent déclencher la même procédure, ce n'est pas du tout un souci. Il suffit de séparer tous les événements concernés par des virgules après le mot clé **Handles**. Si, par exemple la remise à zéro doit être effectuée en cas de clic sur le bouton **Résultat**, mais aussi de clic sur le bouton **Annulation**, on aura :

```
Private Sub RemiseAZero() Handles Résultat.Click, Annulation.Click
    instructions
End Sub
```

Et voilà le travail.

En réalité, il y a juste une petite subtilité supplémentaire. C'est que lorsqu'un événement déclenche une procédure, Visual Basic prévoit automatiquement un passage de paramètres depuis l'événement jusqu'à la procédure, passage de paramètres sans lequel on serait parfois bien embêté.

La syntaxe complète d'une procédure événementielle sera donc typiquement :

```
Private Sub RemiseAZero(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
Résultat.Click, Annulation.Click
    instructions
End Sub
```

Où, à chaque exécution de la procédure suite à un événement : :

sender est un paramètre en entrée (par valeur) qui désigne l'objet qui a déclenché l'événement

e représente un paramètre en entrée (par valeur) qui spécifie les conditions de l'événement.

Gardons ces généralités en tête ; nous aurons naturellement l'occasion de revenir plus en détail sur ces deux paramètres et la manière de s'en servir.

Si vous avez compris ce second mécanisme que sont les procédures événementielles, alors vous avez alors pigé 45 % supplémentaires du cours. Donc, avec les 45 % précédents, on en est à 90 %. Tout le reste, c'est de la petite bière. Puisque je vous le dis.

Visual Basic .NET

Partie 2

L'interface de Visual Studio

Après avoir fait connaissance des concepts de la programmation objet, nous allons dans ce chapitre nous familiariser avec l'interface VB, c'est-à-dire avec les principaux outils que nous offre l'environnement de développement Visual Studio. "*Oui, mais c'est quand qu'on commence à programmer ?*" Ca vient, ça vient, patience...

1. Structure des applications

La première chose à faire, c'est d'assimiler l'architecture des édifices que nous allons créer et le vocabulaire qui va avec. Et ça, ce n'est pas rien, tant à côté d'une application VB, les plans d'une usine à gaz sont d'une simplicité biblique. Mais enfin, pas le choix, et autant consacrer quelques minutes à se mettre les idées au clair plutôt que perdre ensuite des heures juste parce qu'on s'est mélangé les pinceaux dans les arborescences et les dénominations.

Alors, une application VB.NET, c'est quoi ? C'est :

- un ensemble de fichiers formant ce qu'on appelle le **code source**, écrit dans le langage Visual Basic.
- un **fichier exécutable**, produit à partir de ce code source. Rappelons que le fait d'engendrer un fichier exécutable à partir du code source s'appelle la **compilation**.

En ce qui concerne le code source, VB va l'organiser de la manière suivante :

le point de départ d'une application VB, c'est une **solution**. Lorsqu'on crée une nouvelle solution, VB demande pour celle-ci un **nom** et un **répertoire**. Aussi sec, il va alors faire trois choses :

1. créer à l'emplacement indiqué le **répertoire** portant le nom de la solution.
2. dans ce répertoire, toujours avec le nom de la solution, **créer un fichier *.sln** contenant les informations de la solution.
3. et toujours dans ce répertoire, toujours avec le nom de la solution, **créer un fichier caché *.suo** , contenant les options choisies dans Visual Basic par le programmeur qui a développé la solution.

Mais ce n'est qu'un combat, continuons le début. En effet, dans nos solutions Visual Basic, nous allons devoir insérer nos applications, c'est-à-dire nos **projets**. Si on le souhaite, une même solution peut contenir plusieurs, voire des dizaines de projets. Cela dit, **je ne recommande pas cette stratégie**. Nous considèrerons donc, en tant que débutants, qu'une application VB = un projet = une solution.

Il va toutefois falloir procéder avec doigté si on ne veut pas que notre disque dur devienne rapidement un foutoir innommable. En effet, chaque projet est lui aussi caractérisé par un **nom** et un **répertoire** de sauvegarde. Mais, et c'est là que c'est fort, **ce répertoire n'est pas forcément un sous-répertoire de la solution qui contient le projet** - même si lorsqu'on débute, ce soit très vigoureusement recommandé.

Il n'y a donc pas de rapport entre le fait que les objets "solution" contiennent les objets "projets", et le fait que le répertoire où est enregistrée la solution contienne les répertoires où sont enregistrés les projets...

Mal à la tête ? C'est l'effet Petitmou. Toujours est-il que VB va non seulement créer un répertoire au nom du projet, mais qu'il va mettre dans ce répertoire toute une série de fichiers, dont un fichier au nom du projet et à extension ***.vbproj**, fichier dans lequel est stockée toute la structure du projet.

Remarque fort utile :

Il faut faire très attention lorsqu'on incorpore un projet dans une solution : en effet, la commande **Nouveau projet** entraîne obligatoirement la création d'une nouvelle solution. En revanche, pour insérer un projet dans une solution existante, il faut choisir **Ajouter un projet**.

Enfin, nous voilà arrivés au bout de notre quête : dans chaque projet, il y a un certain nombre d'éléments de base. Ces éléments sont, pour l'essentiel, des **Form**, ou **formulaires**. Une application windows basique compte un seul formulaire, et une application complexe peut en rassembler plusieurs dizaines. **Chaque formulaire sera sauvegardé dans un fichier différent, dont l'extension sera *.vb**. Il faut noter que c'est dans ces fichiers *.vb que se trouve le code proprement dit, celui-là même que vous allez concevoir à la sueur de votre front altier et entrer dans la machine de vos petits doigts graciles. Alors, autant y faire attention et veiller à ce que les sauvegardes soient faites correctement.

Conclusion : quelques conseils de base

Les débutants que nous sommes appliqueront donc impérativement les règles suivantes :

- une solution par projet, un projet par solution
- le projet devra être sauvegardé dans un sous-répertoire de la solution

En ce qui concerne l'exécutable, il y a quelques petites choses à dire. A la différence de certains autres langages, VB.NET ne produit pas un code directement exécutable par toutes les machines. Les instructions contenues dans un exécutable VB.NET sont écrites dans un langage appelé **MSIL**, pour MicroSoft Intermediate Langage. Comme son nom l'indique, ce langage s'adresse à des couches "intermédiaires" de la machine, à

savoir d'une part Windows lui-même, d'autre part à une couche logicielle poétiquement appelée le **Framework .NET** (qu'on peut traduire par "infrastructure .NET").

Pour pouvoir exécuter un exécutable VB.NET, une machine doit donc impérativement comporter ce Framework.NET, que l'on peut télécharger gratuitement sur le site de Microsoft. Encore heureux.

Comment déclencher la compilation ? En passant par la commande **Générer**, qui propose le(s) nom(s) des projets de la solution encours. Le fichier exe est alors généré dans le sous-répertoire **bin** correspondant au projet.

Remarque utilitaire :

Lorsqu'on écrit un programme, point n'est besoin de générer un exécutable en bonne et due forme à chaque fois que l'on souhaite tester le résultat provisoire de notre travail. VB offre la possibilité de lancer l'exécution d'une application directement à partir de la source.
En réalité, il procède dans ce cas à une compilation rapide, sans nous embêter avec des questions superflues, et exécute le produit de cette compilation dans la foulée.
Cela permet de différer la réalisation de la compilation définitive au moment où le programme aura été testé, et certifié sans erreurs par vos bons soins !

Pour améliorer les petits détails liés au fichier exécutable et à la compilation, on y reviendra... plus tard.

2. Prise en main de Visual Basic

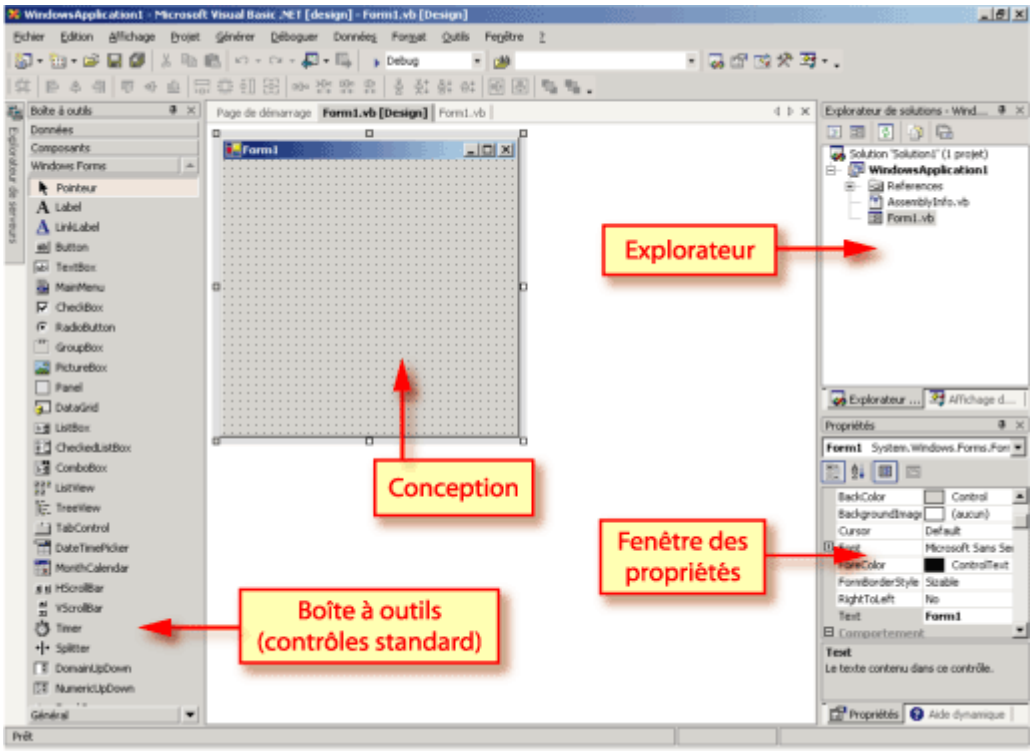
Le logiciel Visual Basic Studio est là pour nous faciliter l'écriture de programmes, en particulier (mais pas seulement) en mettant à notre disposition, sous une forme facilement accessible, les classes Windows les plus communes (boutons, listes, cases, et tout ce qui s'essuie).

2.1 Les deux fenêtres principales

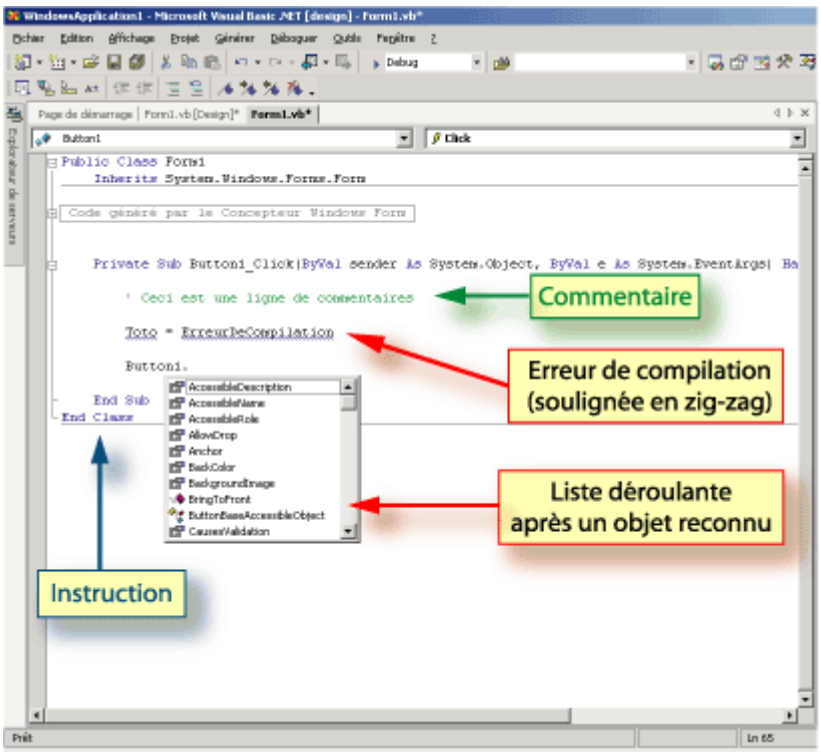
Lorsqu'on va la programmer via Visual Studio, une application va donc toujours pouvoir être abordée sous deux angles complémentaires :

l'aspect graphique, visuel, bref, son interface. Dans la **fenêtre principale** de VB, nous pourrons facilement aller piocher les différents objets que nous voulons voir figurer dans notre application, les poser sur notre formulaire, modifier leurs propriétés par défaut, etc :

Visual Basic .NET



le code proprement dit, où nous allons entrer les différentes procédures en rapport avec le formulaire en question :



Bâtir une application VB, c'est généralement faire de fréquents allers-retours entre les deux représentations. Pour cela, rien de plus facile : il suffit de se servir des **onglets** disponibles sur le haut de chaque fenêtre.

2.2 Créer des contrôles à partir des classes standard

Définition :

Un **contrôle** est un objet créé à partir de certaines des classes définies dans Windows.
En standard, VB propose plusieurs rubriques au sein de sa boîte à outils, et rien que dans la principale d'entre elles, la rubrique **Windows Forms**, une quarantaine de contrôles, ce qui ouvre déjà bien des horizons.
En réalité, le nombre des contrôles disponibles est donc bien supérieur. Mais nous en reparlerons... plus tard.

Créer des contrôles à partir des classes proposées en standard est extrêmement simple. Il suffit d'aller piocher d'un clic de souris le contrôle voulu dans la boîte à outils (qui, dans la configuration normale, se trouve à gauche de l'écran) et d'effectuer un cliquer-glisser sur le formulaire pour donner à ce contrôle la taille et l'emplacement voulus. A signaler, pour les fainéants indécrottables, qu'un simple double-clic dans la boîte à outils constitue une agréable solution alternative.

Par la suite, on peut toujours modifier l'emplacement et la taille d'un contrôle, d'un simple coup de souris bien placé.

Pour supprimer un contrôle, c'est encore plus simple : on le sélectionne, et ensuite la bonne vieille touche SUPPR le désintègrera et le fera retourner au néant.

Si l'on veut manipuler plusieurs contrôles du formulaire à la fois, on peut sélectionner toute une zone (par un habile cliquer-glisser), ou encore sélectionner individuellement chaque contrôle en maintenant la touche CTRL enfoncée. Bref, que du très banal pour les utilisateurs avertis de Windows que vous êtes.

2.3 La fenêtre des propriétés

Chaque fois qu'un contrôle (ou plusieurs) est sélectionné, la **fenêtre des propriétés** (située en standard à droite de l'écran) affiche les valeurs associées à ce contrôle. C'est-à-dire que se mettent à jour la **liste des propriétés** (qui comme on l'a vu varie d'une classe, donc d'un contrôle, à l'autre) et la **valeur de ces propriétés** (qui varie d'un contrôle à l'autre, même lorsqu'ils sont de la même classe).

Les propriétés qui sont affichées là sont les **propriétés par défaut** du contrôle. Ce sont celles qu'aura le contrôle **au moment du lancement de l'application**. Bien sûr, par la suite, il n'est pas exclu que ces propriétés soient modifiées, notamment par des lignes de code (c'est même généralement un des principaux objectifs des lignes de code, quand on y réfléchit bien). Mais encore une fois, pour fixer les propriétés voulues pour chaque contrôle au lancement de l'application, le plus simple est de fixer à la main, sans code, les valeurs désirées dans cette fameuse fenêtre des propriétés.

2.4 L'éditeur de code

Passons au code. Visual Studio, dans sa grande magnanimité, va tâcher de faire au mieux pour nous faciliter la vie. Il va en fait décrypter notre code au fur et à mesure de sa rédaction, et nous donner en temps réel des indications via des codes de couleur, comme on peut le voir (avec de bonnes lunettes) sur l'image ci-dessus. Ainsi :

les titres de procédures seront écrits en **gris**

les mots-clés du langage seront portés en **bleu**

les commentaires seront en **vert**

Et last but not least, toute ligne comportant une faute de syntaxe, ou posant un problème au compilateur, sera immédiatement **soulignée**. Qui plus est, il suffira d'amener le curseur de la souris au-dessus de l'erreur pour que la raison en soit immédiatement affichée dans une bulle d'aide (mais aussi, hélas, dans un vocabulaire souvent incompréhensible, il ne faut pas rêver non plus). On dispose donc d'un premier outil de déboguage, très pratique, nous assurant que la syntaxe de notre code sera toujours irréprochable. Cela dit, comme vous le savez, la syntaxe n'est pas la seule source d'erreurs dans un programme...

Ce n'est pas tout. Lorsque nous tapons dans le code un nom d'objet, et pour peu que ce nom soit reconnu (c'est-à-dire qu'il existe bel et bien un objet de ce nom dans notre application), **sitôt que nous entrons le point qui termine le nom de l'objet, apparaît une liste déroulante avec l'intégralité des propriétés et méthodes disponibles pour cet objet**. Ce qui veut dire, inversement que :

Remarque finaude :

Si lors de la frappe du point qui suit un nom d'objet, aucune liste déroulante n'apparaît, c'est que cet objet n'est pas reconnu par le langage.
Cela signifie soit que le nom a été mal orthographié, soit qu'on s'évertue à faire référence à un objet inexistant... De toutes façons, c'est un signal d'alarme qui dénote un problème !

Enfin, comme nous sommes dans un langage qui est un langage **structuré**, le code est organisé sous forme de **blocs** qui peuvent être déployés ou masqués, pour plus de lisibilité, via les petites icônes placés à gauche de la tête de chaque bloc. Il ne faut pas hésiter à jouer de cet outil bien pratique pour s'y retrouver dès que l'application commence à grossir.

Visual Basic .NET

Partie 3

Premiers contrôles

Nous en savons à présent suffisamment pour commencer à mettre pour de bon les mains dans le cambouis. Ouvrons donc la boîte à outils, et regardons ça de près. Que trouvons-nous ?

1. La classe Form

La classe **Form** (qu'on appelle également le formulaire), issue de la classe **Windows.Form**, est l'élément de base, obligé et fondamental, de toute application VB pour Windows. C'est sur une Form, et uniquement sur une Form, que nous pourrons éventuellement poser d'autres contrôles. Et même si on peut à la rigueur la rendre invisible (mais il faut être un peu tordu), de toutes façons, elle est quand même là. Corollaire de cette proposition, c'est également dans la Form que seront rassemblées toutes les procédures événementielles liées aux contrôles que nous aurons créés sur cette Form.

Nous pouvons d'entrée noter plusieurs propriétés de la classe **Windows.Form** (donc de tous les objets Form créés à partir de cette classe), propriétés que nous retrouverons dans la plupart, sinon dans la totalités des autres contrôles proposés par VB :

- **Name** : il s'agit du **nom de l'objet**, exactement comme une variable porte un nom de variable. La valeur de cette propriété n'est donc pas visible à l'écran : il s'agit d'un nom qui sert **uniquement dans le code**, pour désigner l'objet en question.
- **Text** : une autre propriété quasi-universelle des contrôles est **le texte qui leur est associé à l'écran**. Pour une Form, il s'agira du texte qui figure en haut, dans la barre de titre. Pour un bouton, ce sera le texte écrit dessus, pour une case, le texte qui figure juste à côté, etc. À l'inverse de la propriété **Name**, **Text** ne joue aucun rôle du point de vue du code, et un rôle essentiel du point de vue de l'interface.

Une erreur à ne pas faire :

Il ne faut évidemment pas confondre les propriétés **Name** et **Text**, dont le rôle n'a vraiment rien à voir.

- **Size** : Il s'agit évidemment de la taille, autre propriété partagée par la presque totalité des contrôles. Il faut remarquer que la propriété **Size est d'un type particulier, le type Size , qui est un type structuré composé de deux Integer.** Pour modifier sa valeur par du code, c'est donc un peu sportif : il faut utiliser une variable de type **Size** déjà existante, ou, le plus souvent, la créer à la volée par le constructeur **New**. Pour fixer par exemple la taille de la Form à 500 pixels sur 300, et sachant que la manière la plus simple de désigner la Form dans une des procédures qui lui sont liées est d'employer le mot **Me**, on pourra écrire :

```
Me.Size = New Size(500, 300)
```

Nous reviendrons dès le chapitre suivant sur cette écriture un peu étrange. En attendant, sachons qu'il y a moyen de contourner l'obstacle, puisque plutôt que nous embêter avec ce type **Size** structuré, nous pouvons, en fouinant un peu - découvrir que nous avons directement accès à deux autres propriétés, **Height** et **Width**, beaucoup plus maniables car de type **Integer**. Ainsi, la ligne ci-dessus pourra avantageusement être remplacée par :

```
Me.Width = 500
Me.Height = 300
```

...ce qui n'est tout de même pas plus tordu, loin s'en faut. Cela dit, je reviendrai sur ces types structurés et sur ce genre d'écritures bizarres... dans très peu de temps, c'est promis.

Remarque pinailleuse :

La propriété **Size** désigne les dimensions **extérieures** d'un contrôle. Si l'on désire connaître, ou définir, ses dimensions **intérieures** (la zone "cliente" du contrôle), on dispose de la propriété **Clientsize**, qui possède la même structure.
La différence entre **Size** et **Clientsize** est particulièrement sensible avec les Form, en raison de l'existence de la barre de titre.

- **Visible** : cette petite propriété booléenne rend des services inestimables, puisqu'elle permet de masquer (quand elle vaut **False**) n'importe quel contrôle
- **Tag** : voilà, pour finir, une bien étrange propriété. Tous les contrôles la possèdent, et pourtant, elle ne sert... à rien. Mais c'est là son principal atout : elle va pouvoir servir à tout ce qu'on veut ! Elle va nous permettre d'associer n'importe quelle valeur (de type **String**) à nos contrôles, et d'en faire bon usage. Nous verrons au cours de ces leçons plusieurs occasions dans lesquelles cette propriété s'avèrera fort précieuse.

D'autres propriétés de la classe Form sont propres à cette classe, et ne se retrouvent pas - ou rarement - dans d'autres classes. Il s'agit par exemple de :

- **Startposition** : qui détermine la position de la fenêtre sur l'écran lors du lancement de l'application.
- **BackgroundImage** : qui permet de désigner une image d'arrière-plan pour une Form.
- **FormBorderStyle** : qui détermine le type de bordures utilisé

Côté événements, les Form sont capables d'en recevoir - autrement dit, d'en gérer - un certain nombre. Parmi ceux-ci, les plus importants, pour commencer, sont sans doute **Load** et **Activate**. Je m'y arrête d'autant plus volontiers qu'on a souvent une fâcheuse tendance à les confondre, alors qu'ils sont loin d'être identiques.

Définitions :

L'événement **Activate** correspond au fait que la Form spécifiée devient la fenêtre active. L'événement **Load** correspond au chargement de la fenêtre en mémoire vive.

Dans le cas d'une application qui ne compte qu'une seule Form, les deux événements se confondent. En effet, l'unique Form se charge au lancement de l'application, devenant par la même occasion la Form active, et le restant sans discontinuer jusqu'à la fin de cette application.

Mais dès qu'une application compte plusieurs Form, les choses se passent tout autrement. Toutes les Form vont en effet être chargées au lancement de l'application (déclenchant ainsi les événements **Load**). Mais une seule sera active. Par la suite, chaque fois que l'utilisateur passera d'une Form à l'autre, il redéclenchera l'événement **Activate** pour la Form sur laquelle il vient d'atterrir.

Les événements **Activate** et **Load** se prêtent particulièrement bien à des instructions d'initialisation : c'est là, avant que l'utilisateur ait eu le temps de faire quoi que ce soit, qu'on remet les compteurs à zéro, qu'on remet les cases et les zones de saisie à blanc (ou qu'on y réécrit les valeurs par défaut), etc.

2. La classe Button

Les spécialistes de la langue de Chexpire auront bien sûr reconnu derrière ce nom sibyllin la classe servant à créer les... boutons.

On retrouve pour les contrôles créés à partir de la classe **Button** les propriétés générales déjà évoquées ci-dessus pour les **Form** : **Name**, bien sûr, mais aussi **Text**, **Size**, **Location**, **Visible** et bien d'autres.

Si la classe **Button** nous intéresse, ce n'est pas tant pour ses propriétés (ce n'est pas lui faire injure que de remarquer qu'elles restent somme toute limitées) que pour sa capacité à gérer un certain nombre d'événements, à commencer par le plus fréquent d'entre eux : **Click**. Nous allons donc pouvoir gérer des procédures déclenchées automatiquement par le clic sur un bouton, en suivant simplement les règles de syntaxe exposées précédemment ici-même.

Avec tout ce que vous avez lu jusque là, vous avez largement de quoi commencer à vous dérouiller les pattes (et les neurones).

Pour réussir ces exercices, vous disposez de toutes les connaissances nécessaires.
Les deux versions du **Cache-cache** correspondent au même résultat, obtenu par deux stratégies différentes (soit on a deux boutons et on les masque alternativement, soit on a un seul bouton qu'on déplace).

3. La classe Label

Commençons par évacuer deux considérations essentielles.

1. Oui, le **Label** est bien originaire de Cadix.
2. Et oui aussi, le nom de prédilection pour un **Label** est effectivement Hélène.

Ces points étant désormais réglés, nous pouvons passer aux choses sérieuses et parler brièvement de ce contrôle, qui correspond donc à une simple étiquette posée sur une Form (généralement, pour éclairer l'utilisateur sur tel ou tel point, commenter une image ou une zone, etc.)

L'élément sans doute le plus important à retenir pour les Label est qu'**il ne s'agit pas de zones de saisie pour l'utilisateur**. Ce qui n'empêche pas le contenu d'un **Label** de pouvoir être modifié par le code en cours d'exécution, ni le **Label** de pouvoir recevoir un certain nombre d'événements, dont le **Click**.

Signalons toutefois trois propriétés que possède la classe Label :

Autosize : propriété booléenne qui active (ou désactive) le redimensionnement automatique du Label en fonction de son contenu - ce qui peut permettre par exemple qu'une partie du texte déborde du contrôle, et devienne invisible.

Borderstyle : propriété qui permet de régler le style - ou l'absence - de bordure atour d'un Label.

Textalign : propriété qui règle l'alignement du texte au sein d'un Label (il y a 9 possibilités, en fonction de l'alignement horizontal et vertical)

Et voilà tout ce qu'il y a d'important à dire sur la classe **Label** pour le moment.

4. La classe LinkLabel

Il s'agit d'une sous-classe de la classe **Label**. En termes de programmation objet, on dira que **la classe LinkLabel hérite de la classe Label**. **L'héritage** est un concept très important quand on commence à faire sérieusement de la programmation objet. Et on a beau être contre l'héritage dans la vie en général, il faut reconnaître qu'en programmation, c'est une invention très utile.

On reviendra plus loin et (un peu) plus en détail sur ce qu'est l'héritage, mais on peut profiter de l'occasion que nous donne le **LinkLabel** pour donner une première définition : une classe qui hérite d'une autre (on parle alors d'une classe **fille** qui hérite d'une classe **parente**) possède toutes les propriétés et méthodes de cette classe parente, plus d'autres. En l'occurrence, un contrôle **LinkLabel** possède toutes les propriétés et méthodes d'un contrôle **Label**, avec quelques facultés supplémentaires.

Cette aptitude supplémentaire des **LinkLabel**, c'est de **gérer un lien hypertexte**, qui va permettre d'ouvrir un navigateur et d'aller à l'adresse web indiquée. Par défaut, c'est l'ensemble du texte du **LinkLabel** qui sert de lien hypertexte, mais on peut très bien paramétrer l'affaire pour que seule une portion du texte joue ce rôle. Et si cette portion est réduite à rien, alors le lien est tout simplement désactivé.

Voici donc les principales propriétés propres à la classe **LinkLabel** :

LinkArea : encore une propriété structurée en deux **Integer**. Le premier désigne le caractère à partir duquel commence le lien, le second le nombre de caractères qui compose le lien. Si la deuxième valeur vaut zéro, alors il n'y a aucun lien actif dans le texte du **LinkLabel**. Ne pas oublier la syntaxe un peu exotique liée à ces propriétés structurées :

```
LinkLabel1.LinkArea = New LinkArea(integer1, integer2)
```

LinkBehaviour : propriété qui règle le comportement du soulignement pour le lien (standard, en cas de survol, toujours ou jamais)

Il y a une série d'autres propriétés, liées notamment aux différentes couleurs que doit prendre le lien selon les situations (actif, visité, etc.), mais elles ne devraient vous poser aucun problème.

En revanche, il faut maintenant ajouter ce qui concerne le lancement du navigateur avec l'url correcte. Car en réalité, à proprement parler, la classe **LinkLabel** en est incapable : tout ce qu'elle sait faire, c'est générer un événement **LinkClicked**, qui ne se produit que lorsque le lien est actif et que l'utilisateur a cliqué dessus.

Il nous faudra donc gérer la procédure correspondant à cet événement, en y introduisant l'instruction permettant d'ouvrir le navigateur avec l'url correcte, c'est-à-dire en y entrant le code suivant :

```
System.Diagnostics.Process.Start("url souhaitée")
```

Visual Basic .NET

Partie 4

Les bases du langage

Maintenant que vous êtes brillamment venus à bout de vos premiers exercices en deux (ou peut-être trois) coups de cuiller à pot, nous pouvons à présent faire une petite halte afin d'accumuler quelques munitions en vue des combats futurs. Nous allons donc lister une série d'instructions de base de Visual Basic, de celles qui servent partout, tout le temps, en tout lieu et dans toutes les positions.

Remarque qui vaut son pesant de lignes de code :

Comme tout langage un tant soit peu récent, VB.Net offre une telle gamme d'instructions que nombre d'entre elles sont redondantes.

Rien que pour faire une boucle, par exemple, il y a au moins cinq écritures possibles. Le but de ce cours n'étant pas de remporter des concours d'érudition (surtout en Visual Basic, il y a tout de même des sujets d'étude plus affriolants), nous ferons ici dans l'utilitaire : pour chaque usage, une instruction, rarement deux, et jamais plus.

Quant aux autres instructions possibles, nous les ignorerons superbement.

Et voilà pour elles.

1. Les variables (déclaration, types et portées)

En VB.Net, à moins d'aller trifouiller l'option adéquate, par défaut, **la déclaration de toutes les variables est obligatoire.**

Ce qui nous conduit tout droit vers deux problèmes : celui du choix du type de la variable, et celui de sa **portée** (qu'on a parfois un peu tendance à oublier quand on débute).

1.1 Les types de variables

En ce qui concerne le type, voici le tableau récapitulatif des possibilités offertes par VB.Net :

Type	Plage des valeurs	Taille
Short	-32 768 à 32 767	2 octets
Integer	- 2 147 483 648 à 2 147 483 647	4 octets
Long	-9 223 372 036 854 775 808 à -9 223 372 036 854 775 807 à	8 octets
Single	-3,4028235E+38 à -1,401298E-45 pour les valeurs négatives 1,401298E-45 à 3,4028235E+38 pour les valeurs positives	4 octets
Double	-1,79769313486231570E+308 à -4,94065645841246544E-324 pour les valeurs négatives -4,94065645841246544E-324 à -1,79769313486231570E+308 pour les valeurs positives	8 octets
Decimal	+/- 79 228 162 514 264 337 593 543 950 335 sans chiffre décimal +/- 7,922 816 251 426 433 759 354 395 033 5 avec 28 positions à droite de la virgule	12 octets
Byte	0 à 255	1 octets
Boolean	True / False	2 octets
String	de 0 à environ 230 caractères	
Char	1 caractère	2 octets
Date	01/01/1 à 31/12/9999	8 octets
Object	référence à un objet	4 octets + objet

Cela dit, nous verrons au fur et à mesure de ce cours qu'il existe en réalité une pléthore d'autres types, qui sont des déclinaisons du type objet. En fait, tout objet (donc, tout contrôle) peut être stocké dans une variable du type de cet objet (donc de ce contrôle). On y reviendra...

1.2 Portée des variables

Visual Basic .NET

En ce qui concerne la portée, voilà le topo : selon la "durée de vie" souhaitée pour une variable, on procèdera à la déclaration de cette variable à différents endroits du code. Il existe en théorie quatre, mais en pratique **trois niveaux de portée** pour les variables :

- **Niveau procédure :** la variable est créée à chaque exécution de cette procédure, puis détruite à chaque fois que la procédure se termine. Elle ne conserve donc sa valeur qu'à l'intérieur de cette procédure, et pour une seule exécution de cette procédure. Dans ce cas, **la déclaration doit être faite à l'intérieur de la procédure**, par exemple juste après l'instruction Sub.

```
Sub Maprocedure()
Dim toto as Integer
     End Sub
```

- **Niveau Form :** la variable est créée dès que la Form est utilisée. Elle reste accessible par toutes les procédures contenues dans cette Form et conserve sa valeur tant que la Form est active en mémoire vive. Dans ce cas, **il faut déclarer la variable** n'importe où **en-dehors d'une procédure de la Form** (en pratique, mieux vaut déclarer l'ensemble de ces variables en tête de la Form : les choses étant déjà bien assez compliquées comme cela, inutile d'en rajouter en éparpillant le code un peu partout. A noter que ce qui précède vaut également pour les structures).

```
Dim toto as Integer

Sub MaProcedure()
          ...
     End Sub

Sub MonAutreProcedure()
          ...
     End Sub
```

- **Niveau projet :** la variable est accessible à toutes les procédures du projet, quel que soit la Form, la classe ou la structure dans laquelle se trouvent lesdites procédures. Pour ce faire, **la déclaration doit être faite dans un module, en employant le mot-clé Public**. Un module est un élément de projet, qui comme la Form sera sauvegardé dans un fichier *.vb, mais qui ne peut contenir que du code (donc, pas de contrôles, rien qui soit visible à l'écran).

```
Public toto as Integer
```

Respectez les règles d'hygiène !

N'employez pas des variables de longue portée à tort et à travers. Cela gaspille des ressources ! Le principe qui doit vous guider est celui de l'économie de moyens : on ne doit déclarer les variables au niveau Form ou au niveau projet que lorsque c'est nécessaire, et pour tout dire indispensable.

En ce qui concerne les tableaux, rien de bien sorcier. Il suffira d'ajouter dans la déclaration, au nom de la variable, les traditionnelles parenthèses.

```
Dim MonTableau(15) As Integer
```

Attention, danger !

Pour les indices des tableaux, la numérotation commence à zéro. Ça, on a l'habitude.
Mais ce qui peut surprendre davantage, c'est que **le nombre figurant entre les parenthèses indique la valeur maximale de l'indice**, qui ne correspond donc pas au nombre de cases du tableau, mais à ce nombre moins un.
Ainsi, dans l'exemple ci-dessus, j'ai déclaré un tableau de 16 cases (numérotées donc de 0 à 15).
Suffit de le savoir.

Pour ce qui est des tableaux dynamiques, pas de souci particulier. VB.Net gère très bien des déclarations du genre :

```
Dim MonTableau() As Integer
```

...qui sera suivie à un moment ou à un autre dans l'application par un :

```
Redim MonTableau(N)
```

Là aussi, toutefois, attention ! L'instruction **Redim**, employée sans plus de précision, **écrabouille sauvagement toute valeur qui se trouvait précédemment dans le tableau**, chose qui peut s'avérer fort gênante. Si l'on veut pratiquer un redimensionnement sans perdre les données déjà présentes dans le tableau, il suffit de ne pas oublier l'instruction **Preserve** :

```
Redim Preserve MonTableau(N)
```

Et voilà le travail.

1.3 Déclaration, affectation et écritures étranges

VB.Net est un langage qui permet un certain nombre d'instructions très synthétiques, qui condensent en réalité **en une seule ligne plusieurs instructions distinctes**. L'avantage, c'est que cela permet d'écrire très vite. L'inconvénient, c'est qu'un esprit non averti peut s'y mélanger d'autant plus facilement les crayons.

Avertissons donc.

Traditionnellement, nous avons l'habitude de dissocier la déclaration d'une variable de son affectation (je parle ici des langages civilisés, et pas des trucs de pervers comme le C). Donc, spontanément, nous dirons :

```
Dim Toto as Integer
...
Toto = 12
```

Eh bien, VB.Net nous permet d'effectuer ces deux opérations d'un seul coup d'un seul, en écrivant :

```
Dim Toto as Integer = 12
```

Ce qui revient à procéder, dans la même instruction, à la déclaration et à l'affectation de la variable.

Encore plus fort : ce que nous avons vu au chapitre précédent, à propos de la propriété **Size** (mais qui est vrai de toute propriété correspondant non à un type simple, mais à une structure). Normalement, pour

créer une nouvelle variable structurée **Taille** et l'affecter à la propriété **Size** du contrôle Bouton, nous serions censé écrire :

```
Dim Taille As New Size
Taille.Width = 200
Taille.Height = 100
Bouton.Size = Taille
```

Car **Width** et **Height** sont les deux champs de la structure **Size**. Cela dit, nous savons donc que nous pouvons affecter la variable **Taille** au moment même où nous la déclarons, ce qui donnerait une écriture plus ramassée :

```
Dim Taille As New Size(200, 100)
Bouton.Size = Taille
```

Mais là où c'est encore plus fort, c'est que nous pouvons encore raccourcir l'écriture, en affectant la propriété dans le même élan que nous déclarons et affectons la variable **Taille** (qui du coup, n'a plus besoin d'exister en tant que variable nommée). C'est ce qui donne l'écriture étrange vue au chapitre précédent :

```
Bouton.Size = New Size(200, 100)
```

Et voilà le travail. Cela dit, reculer devant ces écritures pour s'en tenir au découpage traditionnel ne doit certainement pas être considéré comme une faute, ni même comme une maladresse.

2. Les opérateurs

Côté opérateurs numériques, on a évidemment les tartes à la crème :

- **+** : addition
- **-** : soustraction
- ***** : multiplication
- **/** : division

Mais en plus de la "bande des quatre", on dispose de quelques opérateurs à peine plus exotiques, dont entre autres :

- ^ : puissance
- **Mod** : modulo
- \ : division entière

Les opérateurs booléens, pour leur part, ne brillent pas non plus par leur originalité (on ne s'en plaindra pas) :

- **And** : et
- **Or** : ou

- **Xor** : ou exclusif
- **Not** : non

Enfin, l'unique opérateur caractère est le traditionnel :

& : concaténation

Remarque hygiénique :
Pour des raisons évidentes de clarté du code, je déconseille très vivement l'emploi de l'opérateur d'addition **+** comme opérateur de concaténation, bien que celui-ci puisse jouer ce rôle (il suffit pour cela qu'il opère deux chaînes de caractères).

3. Les tests

Là, c'est du cake. La forme universelle du test est la suivante :

```
If expression_booléenne Then
    instructions_si_vrai
    [Else
    instructions_si_faux]
    Endif
```

Évidemment, la partie entre crochets peut être éventuellement omise. Et l'on rencontrera également la forme plus développée (mais tout aussi classique) :

```
If expression_booléenne Then
    instructions_si_vrai
Elseif expression_booléenne Then
    instructions_si_vrai
Elseif expression_booléenne Then
    instructions_si_vrai
Elseif expression_booléenne Then
    instructions_si_vrai
    etc.
    Endif
```

Et voilà : des tests, il n'y a pour ainsi dire rien de plus à dire.

4. Les boucles

Les grands classiques, il n'y a rien de tel. Aussi, après Smoke on the Water, Stairway to Heaven et Sultans of Swing, la maison vous propose la boucle conditionnelle :

```
While expression_booléenne
    instructions
    End While
```

Qui correspond au fameux Tantque... FinTantQue qui a bercé de sa douce mélopée vos longues soirées algorithmiques. Quant à la boucle à compteur, nous aurons sans surprise :

```
For variable = valeur_initiale to valeur_finale [step pas]
    instructions
Next variable
```

Nous aurons l'occasion de découvrir plus tard d'autres instructions de boucles, particulières à VB.Net. Mais pour un bref moment, ces deux-là suffiront à notre bonheur.

5. Les fonctions prédéfinies

Relevez vos sièges, attachez vos ceintures, et saisissez votre sac en plastique. Nous entrons dans une zone de fortes turbulences.

En abordant les fonctions, nous en venons en effet à une considération qui risque de vous dérouter au premier abord. Mais ne pas faire l'impasse sur ce qui va suivre est une manière de continuer à se familiariser peu à peu avec les concepts propres à la programmation objet. Alors, c'est parti pour le grand frisson.

Une particularité de cette programmation objet, c'est que tout finit par devenir un objet, même quand au départ, ce n'était pas particulièrement le cas. Ainsi, mais ce n'est qu'un exemple parmi bien d'autres, **les simples variables ordinaires sont-elles considérées par VB comme des instances des classes (assimilées aux types) qui ont servi à les fabriquer**. Autrement dit, les simples variables sont elles aussi traitées comme des objets, avec leur propriétés et leurs méthodes. Une **variable de type String** est donc, pour VB, également une **instance de la classe String**. Une **variable Integer** est également une **instance de la classe Integer**. Etc.

Or, **String**, **Integer**, et toutes les autres classes formées à partir des types simples, sont dotées, comme toutes les autres classes, d'un certain nombre de **méthodes**. Et ces méthodes ont été forgées de manière à recouvrir les (anciennes) fonctions traditionnelles pour ces variables. De sorte que la plupart des fonctions qui permettaient de manipuler les variables ordinaires existent dorénavant également sous forme de méthodes qui s'appliquent à ces variables (considérées cette fois comme des instances des différentes classes-types).

Ce "doublon" permet au programmeur peu familier de la programmation objet de retrouver à peu près ses petits, et de continuer à utiliser les bonnes vieilles fonctions dont il avait l'habitude. Mais à côté de l'approche traditionnelle, existe donc maintenant une approche "tout objet", qui, sans nul doute, représente l'avenir de cette famille de langages.

Illustrons tout ceci par quelques exemples, car je sens bien que dans le fond, là, près du radiateur, il y en a qui ont décroché et qui roupillent dur.

5.1 Fonctions de chaînes

Mid(*chaîne, nombre1, nombre2*) : est une fonction bien connue qui renvoie l'extrait de chaîne qui commence au caractère numéro nombre1 et d'une longueur de nombre2 caractères. Cette fonction possède un équivalent sous forme de la méthode **Substring**. De sorte que pour extraire "boum" à partir de "Super, ça boume !", nous pouvons indifféremment écrire :

```
Resultat = Mid("Super, ça boume !", 11, 4)
```

ou

```
Resultat = "Super, ça boume !".Substring(11, 4)
```

Naturellement, l'exemple ci-dessus marche aussi bien avec des variables qu'avec des chaînes libellées en clair.

De même, à la traditionnelle fonction **Len(*chaîne*)** qui nous renvoie le nombre de caractères de la chaîne, correspond la méthode **Length**

À la fonction **Instr(*chaîne1, chaîne2*)** qui renvoie la position de chaîne2 dans chaîne 1 (la fonction renvoyant 0 si chaîne2 ne se trouve pas dans chaîne1) correspond la méthode **IndexOf**

À la fonction **LTrim(*chaîne*)** qui supprime tous les espaces situés à gauche dans la chaîne, correspond la méthode **TrimLeft**

À la fonction **RTrim(*chaîne*)** qui supprime tous les espaces situés à droite dans la chaîne, correspond la méthode **TrimRight**

À la fonction **Trim(*chaîne*)** qui supprime tous les espaces situés à droite et à gauche dans la chaîne, correspond la méthode **Trim**.

À la fonction **Ucase(*chaîne*)** qui renvoie la chaîne en majuscules, correspond la méthode **ToUpper**

À la fonction **Lcase(*chaîne*)** qui renvoie la chaîne en minuscules, correspond la méthode **ToLower**

À la fonction **Format(*chaîne, chaîne de format*)** qui permet d'obliger la chaîne (ou un nombre) à respecter un certain format d'affichage, correspond la méthode du même nom, **Format**. Pour ce qui est des spécifications de la chaîne de format, je vous renvoie sur l'aide, parce que sinon, on y est encore au prochain réveillon, et ce soir-là, j'ai prévu autre chose.

Certaines fonctions, toutefois, et pour des raisons parfaitement obscures, n'ont pas été dotées d'équivalent sous forme de méthode. Il s'agit par exemple de :

Chr(*nombre*) qui renvoie le caractère ayant nombre comme code ASCII.

ChrW(nombre) qui renvoie le caractère ayant nombre comme code Unicode (rappel : le code Unicode, à la différence du code ASCII, est composé de deux octets)

Inversement, et de manière tout aussi incompréhensible pour les esprits rationnels, il existe certaines méthodes qui accomplissent des tâches qui ne sont assumées par aucune fonction, telles :

Remove, qui permet de supprimer des caractères dans une chaîne (en précisant à partir de quel caractère et combien on en supprime)

Insert, qui permet d'insérer une chaîne dans une autre à une position donnée.

Split(*caractère*) qui éclate la chaîne en un tableau (de chaînes), en fonction du séparateur indiqué par le paramètre caractère. Ceci peut s'avérer pratique pour "découper" un chemin menant à un fichier, par exemple, et récupérer dans un tableau la liste des répertoires et des sous-répertoires en un tournemain.

Join(*caractère, tableau de chaînes*) qui effectue l'opération inverse : elle constitue une chaîne à partir des chaînes contenues dans le tableau de chaînes passé en paramètre, et en insérant à chaque fois comme séparateur le caractère passé en paramètre.

Il nous reste un dernier problème à traiter avec les chaînes : celui de leur comparaison. En effet, une pelote d'épines classique est de savoir si "bonjour" doit être considéré comme la même chaîne que "BonJour" ou que "BONjoUR". VB nous permet donc de définir une option de comparaison, afin de trancher nous-mêmes dans le lard, et de décider ce qui nous arrange. Nous pouvons donc, une bonne fois pour toutes, régler cette valeur dans les propriétés du projets, par la rubrique **Option Compare** :

Binary signifiera que les caractères seront évalués d'après leur codage binaire. Majuscules et minuscules seront dès lors traitées comme des caractères différents.

Text signifiera à l'inverse que les caractères seront évalués en tant que tels, et que majuscules et minuscules seront considérées comme représentant le même caractère.

Une autre possibilité consistera à utiliser au choix la fonction **StrComp** ou la méthode **String.Compare**, dont nous passerons les deux chaînes à comparer en paramètres, et dont nous fixerons le troisième argument à **False** (si nous voulons une comparaison "binary") ou à **True** (si nous voulons une comparaison "text").

StrComp et **String.Compare** renvoient toutes les deux -1 si la première chaîne est placée avant la seconde dans l'ordre alphabétique, 0 si les deux chaînes sont égales, et 1 si c'est la seconde chaîne qui précède alphabétiquement la première. Cette technique nous permettra de spécifier les options de comparaison non une fois pour toutes, mais à chaque fois, au gré de nos besoins qui peuvent varier au cours d'un même programme.

5.2 Fonctions et méthodes numériques

Là, cela va aller nettement plus vite (personne ne s'en plaindra, et surtout pas moi). Tout d'abord, parce que le nombre de fonctions disponibles et souvent utilisées et nettement moins grand. Mentionnons vite fait bien fait :

CInt : qui est la fonction qui renvoie la partie entière d'un nombre.

Rnd : qui génére un nombre aléatoire compris entre 0 (au sens large) et 1 (au sens strict).

Remarque surprenante :
Visual Basic a du mal à générer de vrais nombres aléatoires. En fait, si l'on ne fait rien, une suite de **Rnd** dans un programme produira invariablement à chaque exécution la même série de nombres aléatoires !
La parade consiste, avant chaque appel de la fonction **Rnd**, à passer l'instruction **Randomize**, toute seule sur une ligne. Cette instruction revient en quelque sorte à donner l'ordre au croupier de battre les cartes avant une distribution...

Je re-signale au passage, et pour en finir avec les fonctions numériques, que la fonction **Format**, déjà vue à propos des chaînes, s'applique également aux nombres.

5.3 Fonctions de conversion

Visual Basic .Net est globalement très arrangeant sur les types de variables, et le compilateur procèdera de lui-même aux conversions nécessaires chaque fois que le problème se posera. Toutefois, il existe des situations dans lesquelles le programmeur doit effectuer cette conversion, sous peine de voir le compilateur prendre à sa place la mauvaise décision. Citons donc en particulier :

Val(*chaîne*) : qui convertit les caractères d'une chaîne en nombre. Ainsi, Val("547") deviendra 547. Dans le cas d'une chaîne ne correspondant à aucun nombre, Val renverra zéro. Problème, dès que Val tombe sur un point, un virgule, ou quoi que ce soit du genre, il s'arrête et ne prend que la partie entière. D'où l'intérêt des autres fonctions de conversions, du genre :

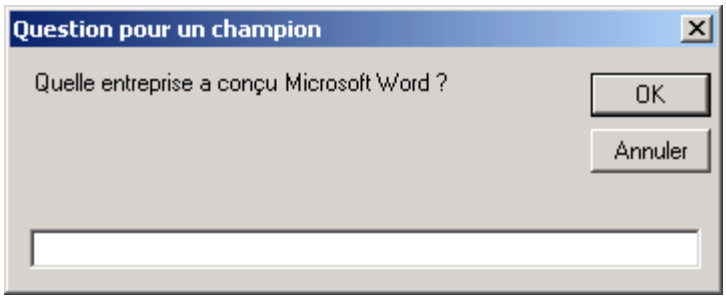
CDbl(*chaîne*) qui convertit une chaîne en réel double. Cette famille de fonctions "*C quelque chose*" existe pour tous les types de données (il y en a toute une série dont **CDbl**, **CBool**, **CByte**, etc.)

5.4 Fonctions d'interface

La fonction InputBox

Nous allons à présent découvrir deux nouvelles fonctions, qui n'ont aucun équivalent dans les langages traditionnels (procéduraux), car elles sont directement liées au fait que VB soit un langage basé sur l'interface Windows.

La première de ces deux fonctions est celle qui nous permet de faire apparaître une petite boîte de saisie tout ce qu'il y a de plus ordinaire, comportant une zone dans laquelle l'utilisateur pourra entrer une valeur :



Je le répète, il s'agit là d'une fonction, dont les deux arguments de type **String** correspondent respectivement à l'invite ("*Quelle entreprise...*") et au titre ("*Question pour un champion*") de la boîte de saisie. Cette fonction renverra toujours une valeur de type **String**. Pour obtenir le résultat ci-dessus, on aura donc écrit un code du genre :

```
a = "Quelle entreprise a conçu Microsoft Word ?"
b = "Question pour un champion"
Rep = Inputbox(a, b)
```

Ce code attendra que le bouton **OK** (ou **Annuler**) soit cliqué, et stockera alors le contenu de la zone de saisie dans la variable **Rep**.

Remarque triviale, mais on n'est jamais trop prudent

Il serait donc aberrant d'entrer sur une ligne, toute seule, InputBox(a, b).
Cela reviendrait à traiter InputBox comme s'il s'agissait d'une instruction, alors qu'elle est, je le répète, une fonction, c'est-à-dire une valeur.

La fonction MsgBox

La fonction jumelle d'**Inputbox**, vouée à un usage un peu différent, est la fonction **MsgBox**. Elle déclenche l'apparition d'une boîte dépourvue de zone de saisie, mais dotée d'un jeu de boutons, et éventuellement d'une illustration standard, du genre :

Là, on retrouve comme arguments l'invite ("*Missile thermonucléaire...*") et le titre ("*Dernière vérification*"). Mais entre ces deux paramètres, on doit fournir un argument d'un genre un peu spécial, sur lequel je vais m'arrêter à présent.

Le rôle de cet argument est double : il doit indiquer d'une part quel est le jeu de boutons souhaité, parmi les six disponibles ("**Abandonner, Réessayer, Ignorer**", "**OK**", "**OK, Annuler**", "**Réessayer, Annuler**", "**Oui, Non**", "**Oui, Non, Annuler**"). Il doit également stipuler quelle illustration viendra éventuellement égayer votre boîte à messages, parmi les quatre possibles (**Critical**, **Exclamation**, **Information**, **Question**). Il est possible d'activer de surcroît quelques autres options, mais nous les laisserons de côté provisoirement. Ce qui compte, c'est de comprendre comment ça marche.

En fait, l'argument attendu est de type **Integer**. Et chacune des possibilités de boutons et d'illustrations énumérées ci-dessus est associée par VB à une valeur entière. Dès lors, pour choisir tel jeu de bouton, il suffit d'entrer comme argument l'entier correspondant. Et pour associer tel jeu de bouton à telle illustration, il suffira d'additionner les deux valeurs correspondantes. Miracle de l'intelligence petitmollienne, les valeurs ont précisément été choisies pour que toutes les additions donnent des résultats différents, et que la machine puisse s'y retrouver sans ambiguïté possible.

Là où ça devient intéressant, c'est qu'on n'est pas obligé de spécifier l'argument sous la forme d'un nombre entier, laborieux à aller chercher pour celui qui écrit le programme, et encore plus laborieux à comprendre pour celui qui le relit. Cet entier peut donc être écrit sous une forme, celle d'une **constante VB**.

Définition simple :

Une **constante VB** est un mot-clé du langage, qui pour le compilateur, équivaut à un entier donné.

Définition savante :

Une **constante VB** est en réalité **le membre d'une énumération**.

Vous ne comprenez rien à cette définition ? C'est normal. Je l'ai mise là juste pour pouvoir vous dire dans quelques chapitres que je vous en avais déjà parlé.

Oui, je ne suis qu'un ignoble manipulateur.

Toujours est-il que la boîte de message figurant plus haut peut ainsi être programmée comme :

```
a = "Missile thermonucléaire activé. Voulez-vous continuer ?"
                b = "Dernière vérification"
                 rep = MsgBox(a, 3 + 32, b)
```

Ce qui n'est pas très explicite. Mais on peut tout aussi bien - et même mieux - écrire :

```
a = "Missile thermonucléaire activé. Voulez-vous continuer ?"
                b = "Dernière vérification"
rep = MsgBox(a, MsgBoxStyle.YesNoCancel + MsgBoxStyle.Question, b)
```

Histoire de faire simple, les types de chez Bill ont en fait donné deux séries de noms différents aux mêmes constantes. En fait, pour désigner exactement les mêmes valeurs **Integer**, on dispose :

d'une part des noms généraux du Framework .NET (ce sont les **membres des énumérations**)

d'autre part des noms propres à VB (ce sont les **constantes VB**).

Résultat des courses, on peut tout aussi bien employer les les uns que les autres, et écrire :

```
a = "Missile thermonucléaire activé. Voulez-vous continuer ?"
b = "Dernière vérification"
rep = MsgBox(a, vbYesNoCancel + vbQuestion, b)
```

Passons maintenant au résultat de la fonction **Msgbox**. Quelle valeur celle-ci renvoie-t-elle ? **Le résultat d'une fonction Msgbox, c'est le bouton sur lequel l'utilisateur a appuyé pour la faire disparaître**. Et c'est ce résultat qui sera stocké dans la variable **Rep** de mes exemples.

Sous quelle forme ce résultat s'exprime-t-il ? Eh bien toujours pareil, sous la triple forme d'un entier correspondant au bouton en question, d'une **constante VB** correspondant à cet entier, ou du **membre de l'énumération** .NET correspondant tout à la fois à l'integer et à la constante VB.

Un petit coup d'oeil sur l'aide nous apprend ainsi que le clic sur le bouton **OK** correspond au renvoi par **MsgBox** de la valeur 1, autrement dit du membre **MsgBoxResult.OK** ou de la constante VB **vbOK**. De même, le clic sur le bouton **Annuler** correspond au renvoi par **MsgBox** de la valeur 2, autrement dit du membre **MsgboxResult.Cancel** ou de la constante VB **vbCancel**. Et voilà le travail.

5.5 Les espaces de noms

Nom de nom. Qu'est-ce que c'est encore que ce truc ?

On vient de voir qu'en VB.Net, la plupart des fonctions existent également sous forme de méthodes des classes présentes dans le Framework .Net. Pire, un certain nombre de fonctions (qui n'en sont plus, du coup) n'existent que sous cette forme. C'est un premier point.

Deuxième point, les classes du Framework .Net ne se situent pas toutes au même niveau. Ces classes sont hiérarchisées, un peu comme dans une arborescence (pensez aux répertoires et aux sous-répertoires). Ainsi, il y a quelques classes très générales (avec leur méthodes). Puis il y a des sous-classes de ces classes (avec leur méthodes à elles). Puis il y a des sous-classes de ces sous-classes de ces classes, avec leurs propres méthodes. Et ainsi de suite.

Dans ces conditions, si l'on veut utiliser une méthode d'une sous-classe d'une sous-classe d'une classe, on est censé écrire de la manière suivante :

```
classe.sous-classe.sous-classe.méthode
```

Et encore, là, je ne vous le fais qu'avec trois étages de classes. Mais il peut hélas y en avoir davantage. Alors, il existe un moyen d'abréger (et d'alléger) un peu l'écriture : ce sont les fameux espaces de noms.

L'astuce va consister à s'adresser au programme de cette manière : "*voilà, au cours des instructions qui vont suivre j'ai eu la flemme de te donner à chaque fois la série complète des classes et des sous-classes qui mènent aux méthodes que j'ai utilisées. Alors, je te donne une petite liste des classes et sous-classes dont je vais me servir. Chaque fois que tu trouveras une nom de classe et une méthode que tu ne connais pas, fouille donc dans cette liste pour voir si elles ne s'y trouvent pas. Et dis à ton compilateur de me foutre la paix.*"

Prenons un exemple que nous retrouverons par la suite. Pour connaître la liste des sous-répertoires d'un répertoire donné, l'aide nous apprend que nous pouvons avoir recours à la méthode **GetDirectories**, qui s'applique à la classe **Directory**, elle-même dérivée de la classe **IO**, elle-même membre de la classe **System**. Du coup, on est censé se cogner l'écriture :

```
toto = System.IO.Directory.GetDirectories("C:\")
```

Alors, autre solution : on indique, en tout début de programme, avant même toute procédure, que l'on entend aller chercher des choses dans l'espace de noms (c'est-à-dire dans la classe) **System.IO** :

```
Imports System.IO
```

Ce qui nous permet ensuite, à chaque fois que nous en aurons besoin, d'écrire simplement :

```
toto = Directory.GetDirectories("C:\")
```

Savoir manier les espaces de noms peut donc se révéler diantrement utile pour s'épargner de longues recherches. Cette technique n'est donc pas à proprement parler indispensable, mais mieux vaut la connaître.

Partie 5

D'autres contrôles

Nous pouvons maintenant reprendre le cours de nos pensées et nous livrer à notre activité favorite : farfouiller dans la boîte à outils pour en sortir tous les jolis jouets qu'elle contient.

1. La classe Textbox

La **Textbox**, c'est par définition la zone de prédilection pour que l'utilisateur puisse saisir des informations. Il n'y a en fait pas grand chose à en dire, sinon que la **Textbox** partage avec les autre classes la plupart des propriétés standard dont nous avons déjà parlé à plusieurs reprises.

Un détail qui a toutefois son importance : la propriété **Text** d'une **Textbox** désigne ce qui se trouve **à l'intérieur** de la **Textbox**. Autrement dit, ce qu'aura saisi l'utilisateur. Par conséquent, nous pouvons en déduire que tout ce que saisit l'utilisateur via une **Textbox**, même lorsqu'il s'agit de nombres, est traité par VB comme une chaîne de caractères. Naturellement, dans certaines circonstances, des conversions vont s'imposer...

Je mentionne toutefois quelques propriétés typiques de la classe **Textbox**, qui valent le coup d'être signalées :

- **Charactercasing** : qui permet de convertir d'autorité le contenu en majuscules ou en minuscules
- **Multiline** : qui autorise ou non la saisie de plusieurs lignes dans le contrôle
- **Passwordchar** : qui permet de convertir d'office, à l'affichage uniquement, le texte tapé en un caractère (typiquement, l'astérique, pour des mots de passe)
- **Readonly** : propriété booléenne qui interdira éventuellement à l'utilisateur de modifier le contenu de la Textbox.

2. La classe Richtextbox

Il s'agit d'une classe fille de la classe **Textbox**, qui hérite donc de toutes ses propriétés... tout en possédants quelques attributs supplémentaires. Et pas des moindres, puisque les contrôles **RichTextBox**, sont pour leur part capables de gérer un contenu au **format rtf** : c'est-à-dire qu'ils mémorisent de nombreux attributs de formatage, tels que le gras, l'italique, le souligné, l'espacement des paragraphes, et autres fanfreluches du même tonneau.

Naturellement, cette bête est directement prédestinée à gérer des simili-traitements de texte, ou du moins des zones dans lequel l'aspect du texte est important.

Je ne m'étendrai pas davantage sur cette classe, bien qu'elle possède pléthore de fonctionnalités, ou plutôt, justement à cause de cela. Cela nous entraînerait trop longtemps dans des détails techniques qui restent d'une importance mineure quant aux mécanismes essentiels du langage. Et si un jour vous avez besoin d'utiliser une **RichTextBox**, vous trouverez toutes les informations utiles dans la doc.

Alors zou, au suivant.

3. La classe Checkbox

La **Checkbox**, c'est cette charmante petite case carrée que vous connaissez bien :

Passons rapidement sur les propriétés que les **Checkbox** partagent avec la plupart des autres classes, pour en venir à celle qui nous intéresse au premier chef : celle qui nous permet de savoir si une **Checkbox** est cochée ou non (si nous l'utilisons en lecture), ou de faire en sorte que **Checkbox** soit cochée ou non (si nous l'utilisons en écriture). Il s'agit de **Checked** : propriété booléenne, qui vaut naturellement **True** lorsque la case est cochée et **False** lorsqu'elle ne l'est pas.

Là où les choses se compliquent un peu, c'est que dans certains cas, on souhaite pouvoir disposer d'une troisième possibilité : la case "à moitié cochée", indiquant généralement que certains sous-élements correspondant à la **Checkbox** sont actifs, et d'autres non :

Pour pouvoir gérer cette situation difficile, il nous faut avoir recours à deux autres propriétés supplémentaires :

- **Threestate** : propriété booléenne, qui indique, dans le cas où elle vaut **True**, que l'utilisateur a désormais le choix entre trois états et non plus deux.
- **Checkstate** : qui à la différence de **Checked**, fera la différence entre **Checked**, **Unchecked** et **Indeterminate**.

Remarque anticipatrice :

Les **Checkbox** sont parfois rassemblées dans un **conteneur**, qui les présente comme faisant partie d'un même groupe. Cette présentation est purement décorative : elle ne remet jamais en cause le fonctionnement de la **Checkbox**, toujours individuel et indépendant de ses congénères.

Il nous faut enfin mentionner le principal événement associé aux **CheckBox** : il s'agit de **CheckChanged**, qui est déclenché chaque fois que la case est cochée ou décochée.

Remarque pleine de finesses :

On remarquera la différence subtile, mais réelle, entre les événements **CheckChanged** et **Click**, appliqués à une **Checkbox**.

Click ne se produira qu'en cas d'action sur la case via la souris.

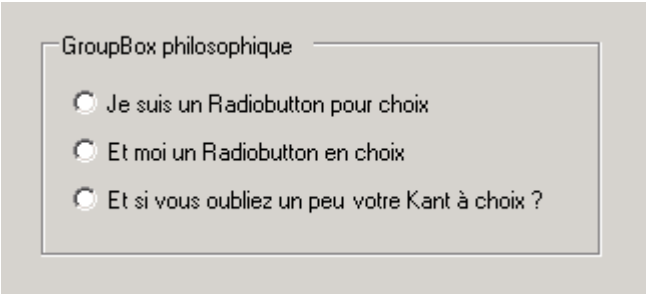
CheckChanged, en revanche, se produira en cas de clic de souris, mais aussi si une ligne de code a modifié l'état de la case. Ou en cas de changement d'état de la case causé par une action au clavier.

Il faut donc réfléchir, avant de choisir l'événement que l'on souhaite gérer, si l'on souhaite réagir ou non à un changement de l'état de la case provoqué par autre chose qu'un clic de souris - si l'application est susceptible de contenir cette autre chose, bien sûr.

4. Les classes Radiobutton, GroupBox et Panel

Trois d'un coup, vous vous rendez compte ? Mais ce ne sont pas les plus pénibles. Voilà tout de suite une illustration de notre propos :

Visual Basic .NET



Le **RadioButton**, dans sa forme classique, n'est donc autre que la célèbre petite case ronde. Par définition, cette case ne fonctionne qu'en groupe, conjointement avec d'autres, puisque par principe, une seule des cases d'un même groupe peut être cochée à la fois. Dit autrement, cocher une case d'un groupe décoche automatiquement les autres cases du groupe.

Ce groupe est défini par ce qu'on appelle un **conteneur**, qui peut au choix être un objet de la classe **GroupBox** ou **Panel**. La différence entre les deux n'est pas furieuse ; disons simplement qu'un **Panel**, 99 fois sur 100, c'est un conteneur sans texte et sans bordure, donc invisible.

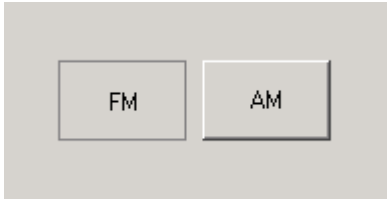
Un **RadioButton** appartient donc au même groupe qu'un autre **Radiobutton** s'il est placé dans le même conteneur (généralement, un **GroupBox**). Ce simple fait suffit à faire marcher les **Radiobutton** les uns avec les autres. Il n'y a pas une seule ligne de code à écrire pour cela.

La propriété la plus utilisée du **Radiobutton** est évidemment **Checked**, propriété booléenne qui renvoie (ou définit) si un bouton radio est coché ou non.

Remarque pas bête :

Lorsqu'une case **Checkbox** déclenche un événement **CheckChanged**, on est généralement obligé de procéder à un test pour savoir si la case vient d'être cochée ou décochée.
En revanche, s'agissant d'un **Radiobutton**, un tel test est parfaitement superflu : si celui-ci vient d'être cliqué, autrement dit s'il a déclenché un événement **CheckChanged**, c'est forcément parce que le **RadioButton** vient d'être coché.
Voilà une remarque de bon sens à ne pas oublier si vous voulez vous épargner des lignes de codes inutiles.

Encore une petite chose sur les **RadioButton**. Ceux-ci peuvent prendre parfois une drôle de tête, comme dans cet exemple :



Ici, nous avons un poste de radio (c'est de circonstance), où nous pouvons, via deux boutons, choisir une gamme de fréquences ou une autre. Sur l'image ci-dessus, c'est la FM qui est enclenchée.

Eh bien, ce look un peu inattendu est obtenu simplement en remplaçant le **GroupBox** par un **Panel** (invisible, par définition), et en réglant la propriété **Appearance** de nos **RadioButton** à **Button** au lieu de **Normal**.

Partie 6

Les collections

Jusqu'à présent, nous avons toujours considéré les contrôles individuellement. Un contrôle, un nom de contrôle, et au moins une ligne de code. Dix contrôles à traiter, au moins dix lignes de code. Cinquante contrôles à traiter, au moins cinquante lignes. Pas moyen d'y couper.

La seule combine qui nous a permis, dans certaines circonstances, de faire une entorse à ce principe, a été de brancher le même événement survenant sur plusieurs contrôles, sur une procédure événementielle

unique, via l'instruction **Handles**. Ceci nous a également permis de manipuler les propriétés du contrôle ayant appelé la procédure, via le paramètre **sender**.

Ce que nous allons voir à présent, c'est comment on peut généraliser les occasions de traiter les contrôles en série, et plus seulement pièce par pièce.

1. La notion de Collection

Dans les versions précédentes de Visual Basic, existait un outil que tout programmeur débutant pouvait maîtriser en quelques coups de cuiller à pot, et qui rendait des services inestimables : les **tableaux de contrôles**. Cela fonctionnait de la même manière que les tableaux de variables, et cela ouvrait les mêmes possibilités.

Bref, c'était tellement simple, tellement intuitif et tellement efficace que dans la dernière version de VB, à savoir VB.Net, celle-là même que vous êtes en train d'étudier, bande de petits veinards, eh bien **les tableaux de contrôles ont tout simplement disparu**.

En lieu et place, nous voilà à présent pourvus d'une série d'ustensiles et d'un bric à brac plus ou moins heureux et maniable, dans lequel nous allons devoir piocher tant bien que mal en fonction des circonstances. Au passage, si quelqu'un de chez Microsoft comprend la logique de cette évolution du langage, et qu'il se sent capable de me l'expliquer, qu'il ne se gêne surtout pas. Il a néanmoins intérêt à être sacrément convaincant, parce qu'autant lui dire tout de suite, ce n'est pas gagné.

Bref, exit les tableaux de contrôles, que nous reste-t-il ? Un machin qui s'appelle les **collections**, machin qui se rapproche furieusement des tableaux de contrôles... sans jamais leur ressembler complètement.

Commençons par dire que par définition, sans même que nous ayons besoin de faire quoi que ce soit pour cela, **tout contrôle posé sur une Form devient aussi sec un membre d'une collection**, en l'occurrence la collection **Controls**. C'est-à-dire que ce contrôle devient le membre d'un ensemble, que nous allons traiter comme tel, et sur lequel nous allons pouvoir programmer des boucles. Car il est possible de balayer l'intégralité des membres d'une collection par une boucle, tout comme on peut balayer l'intégralité des éléments d'un tableau. Sauf que tout n'est pas toujours exactement simple dans cette affaire...

2. Désigner les contrôles par leur indice

La manière la plus évidente de désigner les contrôles dans une collection, va être de se référer à leur **indice** dans cette collection.

"*Faudrait savoir !*" Tel est le cri que j'entends déjà sourdre parmi les masses estudiantines opprimées, suite à la lecture de la phrase précédente. En effet, il y a à peine quelques paragraphes de cela, j'expliquais que les tableaux de contrôles n'existaient plus en VB.Net. Et là, je me ramène avec des collections où les contrôles sont désignés par des indices... Alors, incompétence manifeste ? Absence totale de conscience professionnelle ? Alzheimer précoce ? Schizophrénie caractérisée ?

Rien de tout cela. Dans la collection **Controls**, les contrôles possèdent certes un indice, et cela constitue un point commun avec les tableaux. Mais il reste tout de même de sérieuses différences :

- On ne choisit pas toujours quels contrôles sont membres de la collection : comme on l'a vu, tous les contrôles d'une Form sont d'office membres de la collection **Controls**.
- On ne choisit pas non plus, du moins en mode Design, l'indice attribué à chaque contrôle. Pour info, celui-ci est l''inverse de l'ordre de création des contrôles dans la Form : celui qui a été créé en dernier a l'indice 0, l'avant-dernier l'indice 1, etc. Ce qui s'avère extrêmement déroutant pour le novice.

En l'occurrence, pour balayer les contrôles d'une Form, on pourra donc faire une boucle sur l'indice de ces contrôles dans la collection **Controls**. Le premier indice est par définition 0. Quant au dernier, nous pouvons le déduire de la propriété **Controls.Count**, qui renvoie le nombre de contrôles dans la collection **Controls**. Dans la forme la plus étendue du code, il faudra, pour accéder à l'indice, passer par la propriété **Item**. Le code permettant par exemple de rendre visible tous les contrôles d'une Form sera donc :

```
For i = 0 to Controls.Count - 1
 Controls.Item(i).Visible = True
Next i
```

Le plupart des collections - car je le rappelle, il existe bien d'autres collections que la collection **Controls** - autorisent toutefois l'omission de la propriétés **Item**. On peut donc souvent écrire en abrégé, et pour le même résultat :

```
For i = 0 to Controls.Count - 1
   Controls(i).Visible = True
Next i
```

Remarque instructive :

Tous les contrôles membres d'un contrôle conteneur, tel **Groupbox** ou **Panel**, sont d'office membres de la collection que représente ce conteneur. Une ruse peut donc consister à placer sur la Form un conteneur (au besoin invisible) dans le seul but de pouvoir traiter les contrôles qui s'y trouvent comme une collection.

Remarque perfide :

Les contrôles membres d'un contrôle conteneur ne sont du même coup plus considérés comme membres de la collection **Controls** !

En fait, les contrôles d'une Form sont organisés en quelque sorte dans des collections de différents niveaux. Dit d'une autre manière, les différentes collections d'une Form représentent une véritable arborescence, dont la collection **Controls** n'est que la racine.

Voilà une remarque à méditer longuement.

Bon. Certes. Mais il y a d'autres moyens de parcourir une collection...

3. La boucle For Each ... Next

Cette structure de boucle permet spécifiquement de parcourir une série de contrôles. Tout repose sur le fait que la variable qui va servir de compteur à la boucle n'est ici plus un nombre, comme dans toutes les boucles **For ... Next** que nous avons écrites jusque là, mais que **cette variable représente directement un contrôle**. Autrement dit, la boucle va se servir d'une variable qui va désigner successivement plusieurs contrôles. Donc d'une variable de type **Control**.

Le code précédent pourra ici être réécrit comme :

```
For Each Truc in Controls
   Truc.Visible = False
      Next Truc
```

La variable Truc étant de type **Control**, sa déclaration aura auparavant été :

```
Dim Truc as Control
```

En soi, et sur l'exemple choisi, cette technique n'apporte rien de spécial par rapport à la précédente. Mais comme on le verra plus loin, il y a des circonstances où elle révèlera des avantages notables.

Il faut noter que dans le cas où la collection parcourue ne comporte que des contrôles de la même classe, on peut tout à fait déclarer la variable objet directement comme un membre de la classe en question. Ainsi, au lieu d'avoir un :

```
Dim Truc as Control
```

On pourrait très bien déclarer Truc comme Label :

```
Dim Truc as Label
```

ou comme bouton :

```
Dim Truc as Button
```

...etc.

Encore une fois, cela suppose que tous les contrôles de la collection soient bien de la classe spécifiée. Dans le cas contraire, il y aura forcément une erreur à l'exécution...

4. Tester le type d'un contrôle

Avançant à l'aveuglette parmi les membres divers et variés de la collection **Controls** de la Form (ou parmi toute autre sous-collection, telle celle d'un contrôle conteneur), nous pourrions bien avoir parfois besoin d'un moyen pour identifier ne serait-ce que la classe des ces différents membres. C'est possible grâce à l'instruction

```
Typeof  ... Is
```

Par exemple, le code suivant met à blanc toutes les zones de textes (et uniquement elles) d'une Form :

```
For Each bidule in Controls
If TypeOf bidule is Textbox Then
```

```
            bidule.Text = ""
        Endif
    Next bidule
```

Jouons donc un peu...

Pour **Le maillon faible**, l'intérêt est évidemment de trouver un moyen de programmer la série de questions par une boucle. De sorte que s'il y avait 25 questions et non 5, le code n'en serait pas plus long.

5. Créer ses propres collections par du code

Il y a la collection **Controls**, créée et gérée automatiquement par VB, avec toutes les limites que nous venons de découvrir. Mais il existe aussi la possibilité de **créer ses propres collections par du code**. Bien que cela soit un peu fastidieux, c'est parfois le seul moyen de gérer correctement certains problèmes.

5.1 Déclarer une collection

La première chose à faire sera de déclarer la collection, en lui donnant un nom (exactement comme on déclare une variable, ou un tableau). Par exemple :

```
Dim Ensemble as New Collection()
```

5.2 Remplir et vider une collection

Ensuite, il va falloir entrer dans cette collection, un par un, tous les éléments qui devront en faire partie, via la méthode **Add**. Par exemple :

```
Ensemble.Add(Me.Textbox1)
Ensemble.Add(Me.Textbox2)
Ensemble.Add(Me.Label5)
etc.
```

Disons tout de suite qu'on peut, de même, retirer un contrôle d'une collection par la méthode **Remove** :

```
Ensemble.Remove(Me.Label5)
```

Arrivés à ce stade, nous savons donc gérer comme bon nous semble les membres d'une collection. Cela nous permet notamment de pouvoir procéder sans problèmes à des balayages systématiques (par exemple pour initialiser les valeurs de toute une série de contrôles, pour les rendre visibles ou invisibles, etc.). Cela nous permet également de désigner chaque contrôle membre de notre collection par son indice dans cette collection, exactement comme nous le faisions avec la collection prédéfinie **Controls**. Enfin... presque !

Remarque bonne pour la santé mentale :

Les concepteurs de Visual Basic .Net ont décidé que si les indices des collections prédéfinies, telle **Controls**, commençaient en toute logique à zéro, les indices des collections créées par le programmeur commenceraient quant à eux à 1 !

Si le gars de chez Microsoft dont j'ai parlé tout à l'heure m'appelle, tant qu'il y est, qu'il prépare aussi un argumentaire pour m'expliquer cela. Et qu'il fasse preuve d'imagination...

5.3 "Brancher" les événements sur une procédure unique

Passons. Un autre avantage considérable de disposer de tableaux de contrôles, était de pouvoir traiter un même événement survenant aux différents membres du tableau avec une seule procédure. Dans l'état actuel de nos connaissances, c'est certes possible, via l'instruction **Handles** qui figure dans les titres des procédures : il suffit de faire suivre cette instruction **Handles** de la liste complète des membres de la collection. C'est d'ailleurs cette technique que nous avons déjà eu l'occasion d'utiliser dans plusieurs exercices, sans même pour autant avoir créé de collection. Toutefois, cette combine a des limites :

- dans le cas d'un ensemble formé de très nombreux contrôles, il va s'avérer infernal de devoir énumérer à la main tous ces contrôles après l'instruction **Handles** (pensez au démineur de Windows, dont le plateau de jeu est formé d'un damier de boutons pouvant compter plusieurs centaines d'exemplaires !)
- même si nous n'avons pas encore étudié cette situation, les contrôles ne sont pas toujours créés au départ du programme, via la fenêtre Design. Dans bien des cas, on va souhaiter créer des contrôles par du code, en cours d'exécution. Dès lors, comment rattacher les événements survenant à ces contrôles - qui n'existent pas encore - à telle ou telle procédure événementielle, lorsqu'on écrit celle-ci ?

La parade à ces problèmes tient dans la possibilité de rattacher par du code, donc en cours d'exécution, tel événement survenant à tel contrôle, à telle procédure existante. C'est le sens de l'instruction **AddHandler**, qui donnera par exemple :

```
AddHandler Button2.Click AddressOf MiseAJour
```

Dans cet exemple, **MiseAJour** est le nom de la procédure sur laquelle nous programmons le "branchement" de l'événement **Button2.Click**.

Nous sommes presque au bout de nos peines. Nous savons à présent :

regrouper les contrôles de notre choix dans une collection

parcourir cette collection pour la traiter de manière systématique

associer l'ensemble des contrôles d'une collection à une seule procédure événementielle, afin d'en simplifier la gestion.

5.4 Indentifier le contrôle qui a déclenché l'événement

Il ne nous reste plus qu'un seul souci, mais de taille. Imaginons que nous ayons 125 boutons sur notre Form, sur lequel un clic mène infailliblement à une procédure unique. Comment faire pour récupérer, au sein de cette procédure, le bouton précis qui a provoqué l'événement (ce qui est souvent indispensable) ? Nous disposons certes du paramètre **Sender**, que nous avons déjà utilisé lors d'exercices précédents. **Sender** est une variable objet, qui contiendra le nom du contrôle ayant déclenché l'événement.

Il y a alors de fortes chances pour que nous ayons besoin de savoir à quel indice dans la collection correspond le contrôle désigné par cette variable **Sender**. Pour cela, nous pouvons utiliser la méthode **IndexOf**, qui renvoie l'indice de n'importe quel contrôle au sein de n'importe quelle collection... enfin, presque. J'y reviens dans un instant. Ainsi, imaginons que nous ayons créé un Panel comprenant une série de 100 boutons, pour lesquels une boucle a habilement renvoyé l'événement **Click** vers une procédure **Yaclic**. Supposons enfin que nous souhaitions afficher en titre de la Form l'indice du bouton sur lequel l'utilisateur vient de cliquer. La procédure **Yaclic** aurait alors la tête suivante :

```
Private Sub Yaclic(ByVal sender As System.Object, ByVal e As System.EventArgs)
                                Dim i As Integer
                        i = Panel1.Controls.IndexOf(sender)
                                   Me.Text = i
                                   End Sub
```

Cette technique, simple et efficace, ne fonctionnera toutefois pas au sein des collections que nous aurons créées et remplies par du code, **les collections créées par le programmeur ignorant la méthode IndexOf** ! Pour celles-ci, il faudra avoir recours à une voie détournée : programmer une boucle qui compare, par exemple, la propriété **Name** de chaque membre de la collection **Macollec** avec celui du **sender**, sur le modèle suivant :

```
Private Sub Yaclic(ByVal sender As System.Object, ByVal e As System.EventArgs)
                                Dim i As Integer
                              For i = 1 to Macollec.Count
                        If Macollec.Item(i).Name = sender.Name Then
                                   Me.Text = i
                                     Endif
                                   Next i
                                   End Sub
```

En conclusion : VB.Net permet de faire tout ce que nous permettaient les tableaux de contrôles des versions précédentes de VB, mais au prix d'un code nettement plus compliqué. Et par-dessus le marché, au lieu d'établir une règle simple fonctionnant dans toutes les situations, les gars de chez Microsoft se sont amusés à multiplier les exceptions. Comme quoi, les Shadoks ont incontestablement inspiré Bill Gates lorsqu'ils proclamaient solennellement : "*En essayant continuellement, on finit par réussir. Donc : plus ça rate, plus on a de chance que ça marche*"

6. Créer dynamiquement des contrôles

J'y ai fait rapidement allusion un peu plus haut, alors autant ne pas vous laisser dans les ténèbres de l'ignorance. D'autant qu'une fois que vous aurez appris cette technique, vous aurez à votre disposition un ensemble de mécanismes qui vous permettra de faire face à bien des situations.

Alors voilà, l'affaire est toute bête, et tient en peu de mots : **il est possible de créer et de détruire des contrôles en cours d'exécution, via les instructions appropriées**.

Visual Basic .NET

Créer un nouveau contrôle par du code n'est pas plus compliqué que créer une nouvelle variable : on retrouve le mot-clé **Dim**, et la spécification de la classe du contrôle. Sans omettre le constructeur **New**, indispensable dès que la déclaration porte sur autre chose qu'un type simple. On aura ainsi une ligne du genre :

```
Dim Toto as New Button
```

Où **Toto** sera la propriété **Name** du nouveau bouton créé. Mais à ce stade, nous n'avons créé qu'un bouton désincarné, virtuel, si j'ose dire, et qui n'appartient à aucun ensemble de notre application. Si l'on veut par exemple que le nouveau bouton soit visible (et en général, c'est vivement recommandé), il est indispensable de l'incorporer dans la collection **Controls** de la Form :

```
Controls.Add(Toto)
```

Toto est dorénavant un contrôle comme un autre de la Form. Si, pour libérer de l'espace, on veut à un moment où à un autre détruire le contrôle, il suffira de lui appliquer la méthode **Dispose**.

Remarque pas piquée des hannetons :

Si l'on crée un contrôle dynamiquement, par du code, celui-ci se voit donc déclaré par l'instruction **Dim** au sein d'une procédure. Par conséquent, ce contrôle va être considéré par le langage comme une variable (objet) locale à la procédure. En toute logique, il sera donc impossible de faire référence à ce contrôle par son nom (sa propriété **Name**), dans une autre procédure !

La parade sera d'incorporer, comme nous l'avons fait, ce contrôle à la collection **Controls** de la Form (et éventuellement, à d'autres collections créées par nos soins). De là, nous pourrons accéder au contrôle, depuis n'importe quelle procédure, via son indice dans la collection... Ouf.

Le seul détail qui nous reste à régler - mais par rapport à ce qu'on s'est cogné jusque là, ça va être du cake - c'est de savoir comment on va faire pour attribuer automatiquement des noms (des propriétés **Name**) à toute une série de contrôles créées à la chaîne par du code.

Pour cela, il suffit par exemple de concaténer un nom (mettons, **Toto**), et un nombre généré par une boucle. De toutes façons, l'essentiel est que les noms soient tous différents : le reste, on s'en fiche un peu, puisqu'on ne se resservira des contrôles qu'en les désignant par leur indice dans la collection dans laquelle nous les aurons rangés.

Voilà, pour résumer, un code qui accomplit les tâches suivantes :

il crée et affiche les unes en dessous des autres 20 nouvelles **Checkbox**

il les intègre dans la collection **Mescases**

il branche le changement d'état de chacune de ces cases sur une procédure unique, **ClicMesCases**

```
Dim MesCases As New Collection

Private Sub Button1_Click (ByVal sender As System.Object, ByVal e As System.EventArgs)
                          Handles Button1.Click
   Dim i As Integer
```

```
        For i = 1 To 20
        Dim x As New CheckBox
    x.Name = "Macase" & Str(i)
        Controls.Add(x)
          x.Left = 50
         x.Top = i * 20
          x.Width = 150
    x.Text = "Je suis une case"
        MesCases.Add(x)
AddHandler x.CheckedChanged, AddressOf ClicMesCases
          Next i
        End Sub
```

Pour gérer le clic sur ces cases et afficher laquelle a déclenché l'événement, nous retrouverons donc un code déjà vu, sur un mode un brin laborieux (puisque l'emploi d'une collection personnalisée **MesCases** nous interdit de récupérer directement l'indice via **IndexOf**) :

```
Private Sub ClicMesCases(ByVal sender As System.Object, ByVal e As System.EventArgs)
        Dim i, tut As Integer
    For i = 1 To MesCases.Count
    If MesCases(i).name = sender.name Then
tut = MsgBox("vous avez cliqué sur la case n°" & i, vbOK)
        End If
        Next i
    End Sub
```

Et voilà. Bon, avec ça, on va pouvoir faire des noeuds très jolis (et très gros) à nos neurones.

7. Remarque finale

Si les collections ont beaucoup d'inconvénients, elles possèdent néanmoins un avantage :c'est qu'elles représentent un concept que l'on va retrouver à tous les coins de rues en VB.Net. En fait, si j'ai jusqu'ici parlé uniquement des collections au sens de série de boutons, de labels, etc. c'était parce qu'il fallait bien prendre le problème par un bout. Mais en VB.Net, dès que l'on a affaire à une série d'éléments qui sont "inclus" dans un autre élément, on peut être sûr que le langage considère qu'il s'agit d'une collection. Ainsi, je pourrais citer en vrac :

les éléments d'une liste

les menus, les sous-menus, et les sous-sous menus.

les éléments d'un **Treeview**, avec leurs sous-éléments, leurs sous-sous-éléments.

les différentes Form au sein d'une application MDI (pensez à Word ou Excel, avec les différentes fenêtres "document" au sein de la fenêtre principale de l'application).

etc.

Tout cela, nous en reparlerons. Mais si vous avez compris le concept de collection et les outils avec lesquels on les manie, vous devriez vous en sortir sans trop de dégâts.

Visual Basic .NET

Partie 7

Encore des contrôles (les listes)

Reprenons à présent notre tour d'horizon des différents contrôles proposés par VB.Net. Bon, les **Button**, c'est fait. Les **Label** et les **Textbox**, on les a vues aussi, hmmm... les cases, ça y est... que reste-t-il ? Oh, tiens, ben oui :

1. La classe Listbox

La **Listbox** est à l'ensemble des classes de listes ce qu'est la 2 CV aux automobiles : un prototype ancien et basique, mais solide, rustique, et qu'on a toujours plaisir à voir en service :

La **Listbox**, par définition :

- contient exclusivement des items de type **String**.

- graphiquement, se présente toujours sous une forme "développée" (ce n'est pas une liste déroulante).
- impose un choix parmi les items proposés (ce n'est pas une liste modifiable, où l'on peut entrer une valeur qui n'est pas proposée dans la liste)
- possède ou non une barre de défilement verticale (voire horizontale), selon le nombre d'éléments présents dans la liste et la place disponible pour les afficher (l'apparition et la disparition des barres de défilement sont gérées de manière automatique par VB, même si on peut toujours bidouiller cela via certaines propriétés).

Pour les **Listbox**, on va trouver plusieurs propriétés concernant le mode d'affichage, dont entre autres :

- **Sorted** : propriété booléenne qui indique si la liste est automatiquement triée ou non.
- **Multicolumn** : propriété booléenne qui permet d'afficher les items sur plusieurs colonnes.

Et une propriété fondamentale pour son comportement :

- **SelectionMode** : qui indique si l'on autorise ou non la multisélection des items de la liste.

En ce qui concerne la gestion des items de la **ListBox**, il faut avoir à l'esprit une chose, de laquelle de déduisent toutes les autres :

Théorème :

Un item d'une **Listbox** est une chaîne de caractères, membre de la collection portant le nom de la **Listbox**.

Par conséquent, ce que nous avons appris au chapitre précédent sur les collections s'applique de plein droit aux items des **Listbox**. Comme quoi, on dira ce qu'on voudra, mais au niveau de l'articulation pédagogique, on sent bien que ce cours est drôlement bien goupillé et qu'on a pas affaire à un amateur. Enfin, moi, ce que j'en dis, hein...

Pour balayer les éléments d'une **Listbox** du premier au dernier, nous pourrons donc utiliser leurs numéros d'indice. Attention toutefois, **Listbox** n'autorise pas l'omission de la propriété **Items**, qu'il faudra donc stipuler en toutes lettres. Pour passer l'ensemble des éléments d'une **Listbox** en majuscules, on pourra donc écrire :

```
Dim i As Integer
For i = 0 To ListBox1.Items.Count - 1
ListBox1.Items(i) = ListBox1.Items(i).ToUpper
Next i
```

Remarque de bon ton :

Dans une **Listbox**, les items sont numérotés à partir de l'indice zéro

On pourra également, dans d'autres circonstances, utiliser la boucle **For Each** ... **Next**, par exemple pour récupérer tout le contenu de la liste dans une seule chaîne MaListe :

```
Dim element As Object
Dim MaListe As String
```

```
MaListe = ""
For Each element In ListBox1.Items
    MaListe = MaListe & element
Next element
```

On peut, comme dans n'importe quelle collection, ajouter un élément dans une liste via la méthode **Add** :

```
Listbox1.Items.Add("Nouvel élément")
```

Pour supprimer un élément, on n'a que l'embarras du choix. La méthode **Remove** demandera de fournir l'élément lui-même...

```
Listbox1.Items.Remove("Midnight Jokers")
```

...tandis que la méthode **RemoveAt** demandera un indice :

```
Listbox1.Items.RemoveAt(5)
```

Enfin, **Clear** procèdera au nettoyage complet de la liste, en supprimant tous les éléments d'un seul coup d'un seul :

```
Listbox1.Clear
```

Le but d'une liste, c'est de permettre à l'utilisateur d'en sélectionner un ou plusieurs items. Cette action de l'utilisateur affectera les propriétés :

- **SelectedItem** : qui désigne sous forme de chaîne de caractères l'élément actuellement sélectionné. Si aucun élément n'est sélectionné, cette propriété vaut une chaîne vide.
- **SelectedIndex** : qui désigne par son indice l'élément actuellement sélectionné. Si aucun élément n'est sélectionné, cette propriété vaut -1.

Lorsqu'une sélection multiple est possible sur une **Listbox**, ces deux propriétés renvoient alors des collections, dans lesquelles on peut partir à la pêche pour récupérer les différents éléments.

En ce qui concerne les événements, on peut bien sûr gérer les **Listbox** par l'événement **Click**. Mais un événement propre est disponible, qui ne se déclenche qu'en cas de changement de l'item sélectionné : il s'agit de **SelectedIndexChanged**. Cet événement est à la fois plus restrictif et plus large que le **Click**. Plus restrictif, car il ne se déclenche pas en cas de clic sur un item déjà sélectionné. Plus large, car il détectera un changement de sélection survenant y compris suite à une manoeuvre au clavier.

Bon, c'est pas tout ça, mais on va pouvoir utiliser notre science toute neuve :

Cet exercice n'est pas spécialement facile. Afin de graduer la difficulté, on pourra commencer par en créer une version qui ne se préoccupe pas de la multisélection. Une fois que c'est au point, on introduira la case à cocher, et on modifiera le code là où c'est nécessaire.

2. La classe Combobox

Il s'agit d'une classe-fille à la fois de la classe **Listbox** et de la classe **Textbox**. Elle en hérite donc l'essentiel des propriétés. En gros, on peut considérer qu'une **ComboBox**, c'est une liste modifiable et/ou déroulante :

L'aspect de la **ComboBox** dépend de la valeur donnée à la propriété **DropDownStyle** :

Dropdown : signifie que la zone est modifiable et déroulante

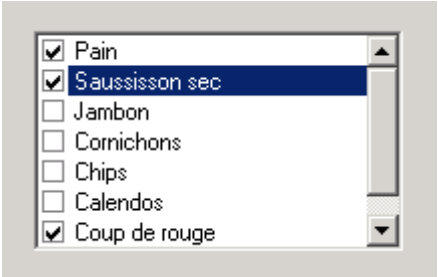
DropDownList : la liste est déroulante, mais non modifiable

Simple : la liste est modifiable, mais non déroulante

Je rappelle que le caractère déroulant d'une **Combobox** n'influence que son aspect, alors que son caractère modifiable influence son comportement (peut-on ou non y entrer autre chose qu'un item de la liste).

3. La classe CheckedListBox

Cette classe est une espèce de croisement entre la **Listbox** et la **Checkbox**. De la première, elle hérite toutes les propriétés hormis celles liées à la multisélection. Cette classe affiche une liste dans laquelle chaque élément est représenté avec une case à cocher :



A noter que :

La propriété **ThreeDCheckBoxes** indique si ces cases sont en relief

l'événement **ItemCheck** est provoqué par le changement d'état (cochage ou décochage) d'un élément

les méthodes **GetItemCheckState** et **SetItemCheckState** permettent respectivement d'accéder et de modifier à l'état d'un élément (parmi **Checked**, **Unchecked** et **Indeterminate**).

les propriétés **CheckedItems** et **CheckedIndices** désignent respectivement les collections **CheckedListBox.CheckedItemCollection** (collection des items) et

CheckedListBox.CheckedIndexCollection (collection des index) des éléments cochés ou indéterminés de la liste.

Comme ça, ça a l'air compliqué, mais c'est plus laborieux que vraiment méchant.

4. La classe ImageList

Ce contrôle n'a aucun intérêt par lui-même. Mais associé à d'autres (que nous verrons dans un instant), il fait un malheur.

Son rôle consiste à être en quelque sorte un **tableau d'images** pour que d'autres contrôles viennent puiser dedans. **Il s'agit d'un contrôle invisible.** Ce n'est pas le seul, mais c'est le premier que nous rencontrons. Encore une fois, il ne sert que d'espace de stockage. D'ailleurs, lorsque que nous le sélectionnons dans la boîte à outils, nous voyons tout de suite qu'il ne se pose pas sur la Form, mais à côté.

On remplit le contrôle **ImageList** par sa propriété **Images**, qui désigne une collection (et à laquelle s'appliquent donc les méthodes **Add** et **Remove**, si on veut en manipuler le contenu par du code). La propriété **ColorDepth** permet de régler la qualité de codage des images en question, et roule Raoul : on accède ensuite aux images que contient **ImageList** par la propriété **ImageIndex**.

Ayé, on a fait le tour de ce petit contrôle.

5. La classe Listview

Il s'agit d'une liste pouvant présenter les informations qu'elle contient de quatre manières possibles, imitant parfaitement le volet droit de l'explorateur Windows (grandes icônes, petites icônes, liste, détails). Ces modes d'affichage sont réglés par la propriété **View**. On peut également faire apparaître une case à cocher devant chaque item via la propriété booléenne **Checkboxes**.

Le maniement de cette classe étant assez particulier, je vous renvoie en cas de besoin à un ouvrage spécialisé (pour l'heure, le courage me manque de rédiger cette partie ; on a beau être enseignant, c'est-à-dire surhomme, on n'en a pas moins ses petits moments de faiblesse).

6. La classe Treeview

Remarque préventive :

La présentation de la classe **Treeview** et des exemples de code qui vont avec constitue indéniablement un des moments les plus pénibles de ce cours qui en compte pourtant beaucoup.
Aussi difficile à lire pour vous qu'elle l'a été à écrire pour moi, cette partie pourra être ignorée par tous ceux qui n'en ont pas un besoin absolu, ou qui n'égayent pas leurs longues soirées d'hiver en s'adonnant aux pratiques masochistes.

Visual Basic .NET

"*Back to the trees !*" criait le pithécanthrope réactionnaire qui refusait la bipédie dans Pourquoi j'ai mangé mon père. Ben, à ce qu'il semble, après quelques millions d'années d'évolution ayant culminé dans le VB.Net, il est effectivement temps d'y retourner.

Dans les arbres, nous allons pouvoir faire plein de choses. En effet, les arbres sont une manière extrêmement pratique de structurer les données, et au moment même où vous lisez ces lignes, je suis certain que vous avez déjà en tête plusieurs exemples de situations où des données informatiques sont organisées - et peuvent donc être représentées - sous forme d'arbre. L'exemple le plus tartignol - c'est évidemment celui que je choisirai - est celui d'un disque dur, avec ses répertoires contenant d'autres répertoires, qui contiennent d'autres répertoires, etc. Et d'ailleurs, si vous voulez imaginer à quoi ressemble un contrôle **Treeview**, il suffit de penser à la partie gauche de l'explorateur de Windows. Ce n'est pas pour rien.

Alors, le Tree d'un **Treeview**, c'est quoi ? C'est un sac de noeuds, au propre comme au figuré.

Chaque noeud (**Node**, en anglais, objet de la classe **TreeNode**) fait partie d'une collection (**TreeNodeCollection**) des noeuds qui ont le même noeud parent que lui (tous les sous-répertoires immédiats d'un même répertoire).

Dans l'autre sens, chaque noeud pointe sur une collection, celle des noeuds dont il est lui-même le père (tous ses sous-répertoires immédiats).

Manipuler un **Treeview**, cela revient donc à manipuler des collections de noeuds ! Ce qui signifie :

ajouter un noeud dans une collection, via **Add** (qui l'ajoute à la fin) ou **Insert** (qui l'ajoute à l'emplacement spécifié)

supprimer un noeud d'une collection, via les trois méthodes déjà) rencontrées à propos des listes : **Remove**, **RemoveAt** et **Clear**.

Côté événements, les **Treeview** proposent plusieurs possibilités originales. On peut notamment gérer le fait qu'un noeud soit déployé, par les événements **BeforeExpand** ou **AfterExpand** (selon qu'on veuille agir juste avant ou juste après le déploiement du noeud). On peut également gérer le fait qu'un noeud soit replié, avec les événements **BeforeCollapse** et **AfterCollapse** (même différence entre les deux).

Remarque de bon sens :

L'exemple de programmation qui va suivre est pour ainsi dire complètement idiot, car comme nous le verrons plus loin, il existe un moyen beaucoup plus rapide de parvenir au résultat dont nous allons laborieusement accoucher ici.

Mais, et les mauvais esprits n'ont qu'à bien se tenir, cet exemple possède une vertu pédagogique, dans la mesure où il nous permettra de mettre le doigt dans des techniques un peu... euh, disons... stimulantes.

Bon, on sait maintenant presque assez pour s'amuser à fabriquer une réplique grandeur nature de la partie gauche de l'explorateur Windows. Pour cela, il nous manque encore quelques petits détails. En particulier, il nous faut un moyen de récupérer, à partir d'un répertoire donné, la liste des sous-répertoires qu'il contient. Ça, ça ne s'invente pas, et c'est pour cela qu'existe la fonction :

```
Directory.GetDirectories(répertoire)
```

qui renvoie, sous forme d'un tableau de chaînes, l'ensemble des sous-répertoires du répertoire passé en argument. Sauf que pour que ça marche, il ne faudra pas oublier d'écrire, tout en haut du programme :

```
Imports System.IO
```

Parce que sinon, le compilateur ne sait pas qui est ce **Directory** dont vous lui parlez, encore moins quelle est cette méthode **GetDirectories** qui lui est attachée, et vous flanquera illico une erreur de compilation. Rappelez-vous, les **espaces de noms**, nous en avons déjà parlé...

Allons-y maintenant pour de bon. Notre code va s'articuler autour d'une petite procédure que nous appellerons chaque fois que nous en aurons besoin pour mettre à jour notre **Treeview**. Cette procédure que j'appelle **Explor** :

réclame comme paramètre entrant (**ByVal**) le noeud dans lequel on veut chercher et afficher les sous-répertoires

purge les sous-noeuds existants par un **Clear**

ajoute comme sous-noeud de ce noeud tous les éléments fournis par **GetDirectories** (c'est-à-dire tous les sous-répertoires), via une boucle **For Each**.

La voici, la voilà :

```
Private Sub Explor(ByVal Node As TreeNode)
        Node.Nodes.Clear()
        Dim s As String
    For Each s In Directory.GetDirectories(Node.FullPath)
        Node.Nodes.Add(Path.GetFileName(s))
        Next s
    End Sub
```

Il faut remarquer que s récupère à chaque fois le nom complet du répertoire (avec le chemin). D'où l'emploi de la fonction **Path.GetFileName**, qui purge le chemin et nous restitue le nom du répertoire proprement dit.

Et d'une.

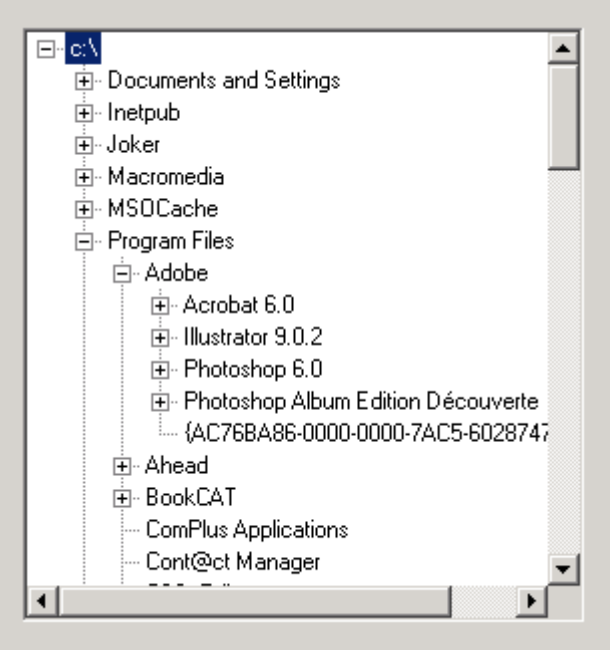
Et de deux, il faut qu'en cas de clic sur un noeud, notre procédure **Explor** soit déclenchée pour chacun des sous-répertoires du noeud en question (on a en quelque sorte à chaque fois un coup d'avance : on affiche dans l'arbre les répertoires situés deux niveaux en-dessous du répertoire actif : comme cela, parmi les sous-répertoires directs du sous-répertoire actif, le **Treeview** affichera différemment ceux qui contiennent quelque chose et ceux qui ne contiennent rien. On aura donc :

```
Private Sub TreeView1_AfterExpand(ByVal sender As Object, ByVal e As
System.Windows.Forms.TreeViewEventArgs) Handles TreeView1.AfterExpand
        Dim z As TreeNode
    For Each z In e.Node.Nodes
        Explor(z)
        Next z
    End Sub
```

Il ne manque plus qu'un détail : amorcer la pompe lors du chargement de la Form, en donnant comme point de départ la racine du lecteur C:, et en considérant que le premier - et le seul - noeud présent dans notre **Treeview** porte l'index zéro :

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles
                                        MyBase.Load
                                TreeView1.Nodes.Add("c:\")
                                Explor(TreeView1.Nodes.Item(0))
                                        End Sub
```

Avec tout cela, on obtient un joli explorateur qui fonctionne, en nous donnant un truc dans ce genre :



Remarque pertinente :

Bien évidemment, on pourrait programmer l'affaire très différemment, par exemple en remplissant d'avance, une bonne fois pour toutes, le **Treeview** avec tous les répertoires du disque dur. Mais ça ne serait pas plus facile à programmer, et cela s'avèrerait en réalité moins performant.

Pour que notre explorateur soit parfait, nickel-chrome comme ceux vus à la TV, il lui manque toutefois un petit détail : qu'à chaque noeud soit associé non seulement le petit "+" ou la simple ligne, mais également l'icône du répertoire ouvert ou fermé, selon les circonstances.

C'est ici que le contrôle **ImageList**, dont nous avons parlé juste avant, va nous rendre un fier service. Il nous suffit d'aller attraper quelque part deux jolies images, l'une d'un répertoire ouvert, l'autre d'un répertoire fermé, et de les mettre dans un contrôle **ImageList**.

Ensuite, associons notre contrôle **ImageList** au **Treeview**, par la propriété... **ImageList** de ce dernier.

A présent, modifions légèrement notre procédure **Explor**, en ajoutant deux instructions :

```
Private Sub Explor(ByVal Node As TreeNode)
                Node.Nodes.Clear()
```

```
        Dim s As String
        Dim z As TreeNode
    For Each s In Directory.GetDirectories(Node.FullPath)
            z = Node.Nodes.Add(Path.GetFileName(s))
                z.ImageIndex = 1
            z.SelectedImageIndex = 0
                Next s
            End Sub
```

Il ne reste plus qu'à faire de même dans la procédure **Form1_Load**, et le tour est joué :

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles
                                    MyBase.Load
                                Dim z As TreeNode
                            z = TreeView1.Nodes.Add("c:\")
                                Explor(z)
                            z.ImageIndex = 1
                        z.SelectedImageIndex = 0
                                End Sub
```

C'est tout de suite plus classieux, hein ?

Remarque de puriste prudent :

A proprement parler, et pour qui a été élevé dans les canons de l'algorithmique rigoriste, les lignes du genre :

z = Node.Nodes.Add(Path.GetFileName(s))

sont une pure profanation. En effet, elles impliquent qu'on puisse à la fois appliquer une méthode (**Add** en l'occurrence) et affecter une variable, le tout d'un même élan. VB.Net autorise donc ce genre d'écriture hérétique, de même qu'il permet, nous en avons déjà parlé, d'affecter une variable tout en la déclarant On peut même trouver sous la plume de certains programmeurs des lignes proprement sataniques, qui consistent tout à la fois à déclarer une variable, à l'affecter et à appliquer une méthode :

Dim z as TreeNode = Node.Nodes.Add(Path.GetfileName(s))

Alors, ce type d'écriture est un peu comme les voitures qui roulent à 250 km/h : c'est en vente libre, et c'est bien pratique pour gagner du temps, mais si on veut rester en vie, on n'est pas obligé de les acheter, et encore moins de s'en servir. A bon entendeur...

Partie 8

Événements insolites

Visual Basic .NET

Il est temps à présent d'examiner plus en détail les différents événements que nous permet de gérer VB. Car il faut bien l'avouer, jusqu'ici, à part des clics, des clics et encore des clics, on n'a pas vu grand chose. Et ce serait bien dommage de s'en tenir là.

1. La notion de Focus

En parlant du **Focus**, ne pas oublier de prononcer le "s" final, sans quoi cela risque de prêter à confusion.

A part ça, le focus, dans une application Windows, désigne le curseur, au sens le plus général du terme. C'est lorsqu'un contrôle possède le "focus" qu'il devient concerné par la frappe d'une touche au clavier (la touche **Entrée** produisant l'enfoncement d'un bouton, par exemple). Selon les contrôles, le focus se matérialise à l'écran par un curseur clignotant (dans une **Textbox**), ou par un liseré sombre (sur un **Button**).

Du point de vue de l'utilisateur, il y a deux moyens de déplacer le focus :

- en cliquant directement avec la souris sur le contrôle désiré (certains contrôles, tels le bouton, ne peuvent toutefois pas recevoir le focus de cette manière, car le clic de la souris y produit directement... un **Click**)
- en appuyant sur la **touche de tabulation**, qui fait circuler le focus d'un contrôle à l'autre.

L'ordre de passage du focus est régi par la propriété **TabIndex** de chaque contrôle : le contrôle qui reçoit le focus par défaut, au lancement de la Form, est celui dont le **TabIndex** vaut zéro. Il va de soi que VB veille à ce que deux contrôles de la même Form ne puissent jamais posséder la même valeur de **TabIndex** (il empêche automatiquement qu'il y ait des doublons, ou des trous dans la numérotation).

Voyons maintenant le point de vue du programmeur. On peut tout aussi bien placer d'autorité le focus sur un contrôle par du code, en lui appliquant la méthode... **Focus**. Celle-ci peut s'employer pour presque tous les contrôles... excepté pour ceux qui ne peuvent recevoir le focus. Étonnant, non ?

Mais le code permet également de détecter l'arrivée du focus sur un contrôle, ou son départ. Il suffit pour cela de gérer respectivement les événements **Gotfocus** et **Lostfocus**, eux aussi disponibles pour la quasi-totalité des contrôles.

2. Les événements clavier

Dans un certain nombre d'applications, on peut souhaiter attribuer certaines conséquences à la frappe de certaines touches du clavier. Par exemple, la touche F1 doit ouvrir le fichier d'aide. Autre exemple, vous pilotez en temps réel les mouvement de **Zorglub**, le grandiose vaisseau de l'hyperespace, grâce aux touches de direction.

Tout cela suppose que la frappe de telle ou telle touche du clavier soit interprétée par le logiciel comme un événement. Aucun problème, Billou s'occupe de nous, et pour ce faire nous offre trois événements, pas un de moins.

- **Keypress** : cet événement détecte le fait qu'un **caractère** a été frappé au clavier.
- **Keydown** et **Keyup** : ces deux événements, qui fonctionnent de pair, se déclenchent lorsqu'une touche du clavier est enfoncée (**Keydown**) ou relâchée (**Keyup**). La caractéristique de ces deux événements est qu'ils détectent **l'état physique** du clavier.

Remarque finaude :
Cela signifie que les touches ne produisant pas de caractères, telles les touches de fonction, ou les touches de direction, ne génèrent pas l'événement **Keypress**, mais génèrent les événements **KeyDown** et **KeyUp**

Revenons-en à présent à un aspect sur lequel, jusqu'à maintenant, nous ne nous sommes pas arrêtés autant qu'il le mérite : je veux parler des paramètres en entrée des procédures événementielles.

Pour ce qui est des généralités sur les paramètres en entrée d'une procédure, je ne vous ferai pas l'affront de vous réexpliquer ce dont il s'agit. Vous êtes blindés en algo, et je sais que jamais vous ne vous seriez lancé dans ce cours sans être des cadors sur la question. Passons donc rapidement, comme il sied de le faire en présence d'experts.

Dans une procédure événementielle, nous avons toujours pu constater, quel que soit l'événement, que VB organisait le passage de deux paramètres : le fameux **Sender**, et le mystérieux **e**.

Fameux, le **Sender**, puisqu'on a déjà vu qu'il s'agissait d'**une variable faisant référence à l'objet qui a déclenché la procédure**. Ceci s'est avéré particulièrement utile lorsque plusieurs objets étaient branchés (par exemple en cas de clic) sur la même procédure : le paramètre **Sender** nous a alors permis, au sein de cette procédure, d'identifier lequel des contrôles avait déclenché la procédure, et d'avoir accès à ses propriétés.

Le paramètre **e** est également une variable qui désigne un objet. Mais cet objet n'est pas le contrôle qui a déclenché la procédure (évidemment, puisque c'est **Sender**, on ne va pas mettre deux fois la même information sous deux noms différents, eh, patate). D'ailleurs, le paramètre objet **e** n'est pas un contrôle du tout. **Il représente, si l'on veut, les conditions, ou les résultats, comme on préfère, de l'événement lui-même.** La nature de ses propriétés variera donc d'un événement à l'autre.

S'il s'agit d'un événement **Click**, disons-le tout net, il n'y a pour ainsi dire aucune propriété dans **e**, car rien n'est plus tristement banal et sans caractéristiques particulières qu'un clic. En revanche, s'il s'agit d'un événement clavier, c'est tout de suite beaucoup plus intéressant.

Par exemple, lors d'un **KeyDown** ou d'un **KeyUp**, **e** possèdera tout un tas de propriétés booléennes (**Shift**, **Alt**) ou numériques (**Keycode**) nous permettant de savoir dans les menus détails quel était l'état du clavier lors du déclenchement de l'événement.

Lors d'un **KeyPress**, nous trouvons pour le paramètre en entrée **e** la propriété caractère **KeyChar**, contenant le caractère généré par la touche pressée.

Ainsi, le code qui afficherait dans une **MessageBox** une à une les touches frappées au clavier serait :

```
Private Sub Button1_KeyPress(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyPressEventArgs) Handles Button1.KeyPress
```

```
        Dim tutu As Integer
    tutu = MsgBox(e.KeyChar, vbOKOnly, "Touche frappée :")
        End Sub
```

Et voilà le travail, enveloppez, c'est pesé.

Il nous reste toutefois un petit détail à régler avant d'en avoir définitivement terminé avec les événements clavier. Imaginons que nous voulions réaliser un "appel à l'aide" avec la touche F1. Pas le choix, nous devrons passer par un **KeyDown** (car la touche F1 n'engendrant aucun caractère, elle ne produit donc pas d'événement Keypress). Mais là où ça coince, c'est quand on réfléchit à quel contrôle nous devrons affecter l'événement. En effet, le focus étant supposé pouvoir se trouver sur bien des endroits de la Form lorsqu'on appuiera sur la touche F1, il faudrait en bonne logique créer une procédure **Chmoll.KeyDown** pour chacun des contrôles **Chmoll** susceptibles de posséder le focus. On n'est pas rendu.

Heureusement, il y a une autre possibilité : faire en sorte que la Form court-circuite tous les contrôles qui se trouvent sur elle en cas de frappe de touche au clavier. Il suffit pour cela de régler sa propriété **KeyPreview** (qu'on pourrait traduire approximativement par "interception du clavier") à **True**. On n'a plus alors qu'à écrire une seule procédure, celle qui gère l'événement **KeyDown** sur la Form. Et dans cette procédure, à tester si la touche frappée était bien F1, auquel cas on déclenche l'ouverture de l'aide. Et hop, comme qui rigole.

3. Événements Souris

Bon, dans la vie d'un informaticien, il n'y a pas que le clavier. Il y a aussi la souris. Et la souris, petit animal vif et malicieux, ça peut faire plein de choses. Ça peut survoler un contrôle. Ça peut se faire appuyer un bouton (ou relâcher un bouton précédemment appuyé) pendant qu'elle est au-dessus d'un contrôle... Bref, une souris, c'est capable d'engendrer une foultitude d'événements aussi intéressants que variés.

Trois de ces événements ne requièrent pas plus de commentaires que cela :

- **MouseHover** : qui détecte le passage de la souris sur un contrôle
- **MouseEnter** : qui détecte l'arrivée de la souris sur un contrôle
- **MouseLeave** : qui détecte la sortie de la souris d'un contrôle

De ces trois événements, qui effectuent le service minimum, et qui n'envoient quasiment aucun paramètre à la procédure qu'ils appellent, il n'y a donc pas grand chose à dire de plus. En revanche, d'autres événements vont nous permettre, via les propriétés du paramètre **e**, de récupérer des tas de renseignements utiles.

- **MouseDown** : événement produit par le fait qu'un bouton de la souris vient d'être enfoncé au-dessus d'un contrôle

- **MouseUp** : événement produit par le fait qu'un bouton de la souris vient d'être relâché au-dessus d'un contrôle
- **MouseMove** : événement produit par le déplacement de la souris au-dessus d'un contrôle

Ces trois événements génèrent donc un objet **e** comportant plusieurs propriétés, dont les plus intéressantes sont :

- **Button** : qui indique quel est le bouton de la souris à l'origine de l'événement, parmi la liste suivante : **Left**, **Right**, **Middle**, **None**.
- **X** et **Y** : qui désignent les coordonnées de la souris **par rapport au contrôle qui reçoit l'événement** (et non par rapport à la Form, attention, piège à... étudiants, et aux autres aussi)

Remarque féline :

A noter qu'un contrôle peut partir à la chasse à la souris, et la "capturer" ! C'est-à-dire qu'il peut capter les événements souris, même s'ils ne se produisent pas au-dessus de lui... Il faut pour cela mettre la propriété **Capture** du contrôle à **True**.

En-dehors des événements qu'elle est capable de produire, la souris est également intéressante dans la mesure où son curseur est un moyen très simple d'informer l'utilisateur de ce qu'il va pouvoir faire avec les différentes zones d'une application. Windows, Word, Excel, entre autres éminents exemples, passent leur temps à triturer le curseur de la souris pour dire à l'utilisateur que là il peut rétrécir une fenêtre, que là il peut élargir une colonne, que là il peut sélectionner toute une ligne, etc.

Savoir manipuler l'apparence du curseur de la souris est donc une chose qui ne coûte pas cher en termes de savoir-faire technique, et qui est très payante pour l'ergonomie d'une application.

Pour commencer, il faut savoir que seul des fichiers de type cursor (***.cur**) ou icone ("***.ico**) peuvent devenir un curseur de souris. Si vous voulez créer des curseurs personnalisés à partir de l'image de votre choix, il faudra donc avant toute chose prendre soin de convertir cette image dans le format adéquat. Une manière simple de procéder est d'utiliser pour cela l'éditeur incorporé dans Visual Studio, auquel vous aurez accès via la commande **Projet - Ajouter un nouvel élément - Fichier curseur**.

Remarque incidente :

Au passage, vous noterez que Visual Studio contient en interne un véritable bric-à-brac d'outils graphiques, permettant de créer et modifier non seulement des curseurs, mais aussi des icônes, et des tas d'autres choses encore.

Il existe toutefois une série de curseurs prédéfinis (les curseurs standard de Windows), qui apparaissent sous la forme de **membres statiques** de la classe **Cursors**. Par exemple, le (trop) célèbre sablier est désigné par **Cursors.Waitcursor**. Nous reviendrons plus loin sur ce qu'est un "membre statique". Mais pour l'instant, contentons-nous de noter que pour modifier l'apparence du curseur au-dessus d'un contrôle, et le transformer, par exemple, dans le (trop) célèbre sablier, on peut - en-dehors du fait de modifier la valeur par défaut de la propriété **Cursor** de ce contrôle - taper le code suivant :

```
Button1.Cursor = Cursors.WaitCursor
```

Dans le cas d'un curseur personnalisé existant sous la forme d'un fichier, on entrera :

```
Button1.Cursor = New Cursor("MonFichierCurseur.cur")
```

Première remarque au cas où :

Lorsqu'on veut modifier un curseur, on n'est pas obligé de modifier le curseur par défaut du contrôle. On peut aussi se contenter de modifier le curseur actuel, à savoir :

Current.Cursor

Deuxième remarque au cas où :

Lorsqu'on charge un fichier pour jouer le rôle de curseur, celui-ci peut être de type ***.cur** ou de type ***.ico**. Cette dernière possibilité est particulièrement intéressante pour produire des effets du meilleur goût (et donner notamment l'illusion qu'on déplace des images alors qu'on ne déplace que le curseur...)

Et voilà le travail, c'est aussi simple que cela. Allez, distrayons-nous un peu :

3. Gérer le Glisser - Déposer (Drag & Drop)

Un des trucs balaises dans l'interface graphique de Windows, c'est qu'on peut se servir de sa souris pour prendre des trucs à un endroit, les trimballer et aller les mettre ailleurs. La quasi-totalité des logiciels exploitent cette possibilité, et il serait quand bien même bien dommage que nous n'apprenions pas à programmer avec VB ce qu'on appelle en français le "Glisser - Déposer", et en anglais le "Drag and Drop". N'est-il pas ?

Cela dit, mieux vaut le savoir, mettre en oeuvre le Drag and Drop, cela suppose une fieffée dose de patience et de rigueur, car le moins qu'on puisse dire, c'est que ça ne glisse pas comme sur des roulettes et qu'à la fin, c'est souvent les armes qu'on dépose. Mais bon, en y allant le plus rationnellement et le plus méthodiquement possible, on peut espérer s'en sortir vivants.

3.1 Approche générale

La première chose à comprendre, c'est qu'un Drag and Drop est constitué de trois événements obligatoires, séparés par une quantité variable d'événements facultatifs. Ces trois événements incontournables se situent au point de départ et au point d'arrivée de la manip :

- au départ, il faut **autoriser (et gérer) le fait qu'un contrôle puisse être alpagué par la souris, et que lui même, ou une de ses propriétés, puisse être trimballé**. Parce que par défaut, aucun contrôle normalement constitué ne peut subir un "Glisser", ou un "Drag". Cela se fait en passant au contrôle la méthode **DoDragDrop**.
- il faut également **autoriser le fait que tel ou tel contrôle puisse recevoir le largage**. Car a priori, aucun contrôle n'accepte de son plein gré d'être la cible d'un "drop". Ceci ne peut se faire qu'en réglant la propriété **AllowDrop** du (des) contrôle(s) cible(s) à **True**.

Remarque :

Ces deux propriétés doivent être modifiées par du code qui, tant qu'à faire, et pour des raisons aisément compréhensibles, sera exécuté juste avant que le Drag & Drop ne se déclenche. Traditionnellement, on considère qu'un bon endroit pour écrire ce code est la procédure **MouseDown** du contrôle qui subit le Drag.

Piège mortel :

Si l'on s'occupe du **DoDragDrop** et du **Allowdrop** dans la procédure **MouseDown**, alors il faut absolument écrire les **Allowdrop** avant le **DoDragDrop**, faute de voir toute l'affaire marcher de manière boiteuse.
C'est étrange et un peu déroutant, mais c'est comme ça.

- à l'arrivée, il faut **programmer ce qui se produit lorsque le bidule que représente le curseur de la souris va être largué au-dessus du (des) contrôle(s) cible(s)**. Et cela se décompose en deux événements : **DragEnter** (entrée du curseur au-dessus du contrôle susceptible de recevoir un Drop) et **DragDrop** (lâchage proprement dit).

Aux trois tâches à accomplir, correspondent donc très logiquement trois événements à gérer. Ce qu'il y a d'hilarant dans l'affaire, c'est que ces trois événements ne correspondent pas aux trois tâches en question ! Un événement (**MouseDown**) s'occupe de deux des tâches, alors que la troisième tâche (la gestion du Drop) mobilise à elle seule deux événements (**DragEnter** et **DragDrop**).

3.2 Les trois événements cruciaux

Résumons-nous, en prenant pour le moment un exemple simple : on va autoriser l'utilisateur à prendre le texte d'un **Label** (que nous appellerons **Etiquette**) pour le poser dans une **TextBox** (que nous appellerons **Arthur**, parce que c'est un joli nom, et que si on l'appelait Perceval, ça risquerait de nous porter la poisse.

Première étape :

Nous devons autoriser **Etiquette** à être l'objet d'un Drag, et **Arthur** à être la cible d'un Drop. Pour le Drag, la méthode **DoDragDrop** devra préciser deux paramètres :

- la nature des informations qui seront transmises à l'objet **e**, objet qui sera créé lors du Drag, qui ne disparaîtra qu'avec le Drop, et dont les propriétés resteront à notre disposition en permanence entre ces deux moments. Je rappelle que **e** n'est autre que ce fameux objet qui figure comme paramètre de certaines procédures événementielles, et dont nous ne nous étions guère préoccupés jusque là. Je rappelle également - au cas où - que si **Sender** représente l'objet qui a déclenché la procédure, **e** incarne quant à lui l'événement lui même, et ses propriétés représentent donc en quelque sorte les circonstances de cet événement.
VB nous demande donc de préciser par ce paramètre, lors du **DoDragDrop**, quelles sont les données qui seront affectées à la propriété **Data** de l'objet **e**.

Ce paramètre a donc parfois une grande importance, et parfois il n'en a aucune : tout dépend, en gros, si plusieurs objets peuvent être à l'origine du Drag, ou s'il n'y en a qu'un seul. Dans le premier cas, nous aurons vraisemblablement besoin de récupérer lors du Drop l'information qui aura été passée en paramètre. Dans le second cas, la récupération de la propriété **Data** de **e** n'est pas utile, et on peut donc affecter à peu près n'importe quoi à **e.Data** !
Considérons ici le cas le plus compliqué : plusieurs contrôles seront susceptibles d'être l'objet d'un Drag. Étant donné que ce qui nous intéresse est le texte du **Label**, nous affecterons à **e.Data** la propriété **Text** d'**Etiquette**.

second paramètre, le type d'effets que le Drag and Drop pourra produire (parmi une liste prédéfinie). Le choix le plus simple est bien entendu **All**. Mais on pourrait d'ores et déjà restreindre les effets possibles en optant pour une des autres possibilités.

Nous entrerons donc, dans la procédure **Etiquette.MouseDown** :

```
Arthur.AllowDrop = True
Etiquette.DoDragDrop(Etiquette.Text, DragDropEffects.All)
```

Deuxième étape :

Nous devons gérer à présent l'entrée du curseur (après un Drag) dans la zone de Drop, c'est-à-dire au-dessus du contrôle **Arthur**. C'est là, dans cette procédure **Arthur.DragEnter**, que nous devrons préciser l'effet qui devra se produire lors du Drop. A noter que cette procédure, et cette instruction, sont indispensables, quand bien même on aura déjà précisé les possibilités lors du **MouseDown**. Si l'on veut, lors du **MouseDown**, on n'a fait que définir ce qui serait possible. Là, il faut dire ce qui va vraiment se passer.

Cela se fait en affectant la propriété **Effects** de l'objet **e**, via une énumération :

```
e.Effects = DragDropEffects.All
```

Troisième étape :

Il ne nous reste plus qu'à préciser, dans la procédure **Arthur.DragDrop**, ce qui doit se passer lors du largage. Ici, c'est très simple : **Arthur** doit prendre le texte qui se trouvait dans **Etiquette**. Si nous avions été sûrs que seul **Etiquette** avait pu être victime d'un Drag, l'affaire auraiot été un peu plus simple. Mais nous avons choisi de traiter le cas général, celui où l'information a été passée lors du **MouseDown** à la propriété **Data** de l'objet **e**.

Récupérer les données trimballées dans la propriété **Data** de **e** n'est pas une mince affaire. On ne peut consulter cette propriété qu'en lui appliquant la méthode **GetData**, méthode exigeant elle-même qu'on précise le format des données à récupérer... données qui doivent être ensuite converties dans le format approprié par la méthode adéquate ! Bref, la simplicité même, dans la plus pure tradition Petitmou.

Dans notre exemple, cela donne :

```
Arthur.Text = e.Data.GetData(DataFormats.Text).ToString
```

Eh oui, quand même, ça calme, hein. Et là, c'est un des exemples les plus simples qu'on puisse imaginer. C'est tout dire.

3.3 Raffinements divers

Ce que nous venons de voir, c'est la base minimale, sans laquelle aucun Drag & Drop n'est possible. Mais on peut tout à fait enrichir l'interface, en tripatouillant la tête du curseur de la souris, en affichant ça ou là des informations, etc. Je me contente ici d'indiquer quelques pistes, tant le sujet est vaste.

Pour commencer, jetons un oeil rapide sur les autres événements qu'il est possible de gérer lors d'un Drag & Drop.

DragLeave : c'est en quelque sorte l'inverse du **DragEnter**. Cet événement se produit lorsque le curseur, pendant un Drag & Drop, quitte le contrôle concerné. Il est particulièrement utile pour gérer le fait que le curseur de la souris sorte du cadre de la fenêtre de l'application (c'est-à-dire de la Form).

DragOver : cet événement est un peu l'équivalent du **MouseMove** en cas de Drag & Drop. Il se déclenche pour tout mouvement de la souris au-dessus du contrôle concerné.

GiveFeedback : cet événement est déclenché dès que le Drag commence (il suit donc, chronologiquement, le **MouseDown** lorsque celui-ci provoque un Drag). **C'est cet événement qui doit être utilisé si l'on souhaite programmer un curseur personnalisé durant le Drag and Drop**.

Ensuite, j'ai parlé du curseur de la souris : il est en effet bon de savoir qu'en cas de besoin, on peut par exemple procéder à une gestion "fine" des coordonnées de la souris durant un Drag & Drop. En effet, chaque événement où la souris est impliquée - et lors d'un Drag & Drop, ils le sont tous ! - envoie au paramètre **e** deux propriétés, **X** et **Y**, qui précisent les coordonnées de la souris au moment du déclenchement de l'événement.

Attention toutefois ! **Selon le type d'événement concerné, ces coordonnées e.X et e.Y sont stipulées par rapport au contrôle qui reçoit l'événement, ou par rapport à l'écran !!!** Autant vous dire qu'il ne va pas falloir s'étonner de certains résultats surréalistes... Dans ce cas, la démarche sera toujours la même : vérifier dans l'aide, et effectuer les conversions de coordonnées nécessaires, comme on aura bientôt l'occasion de le faire dans de croustillants exercices.

La souris communique d'autres informations à l'événement **e**, par exemple l'état de ses boutons - ce qui permet de différencier un clic gauche d'un clic droit.

Enfin, notre bonheur ne serait pas complet si j'omettais de signaler que certains contrôles posent des problèmes particuliers pour le Drag & Drop - et disposent donc de solutions particulières. Il s'agit en particulier des **ListBox**, **Combobox** et autres **Treeview**, qui permettent de Glisser - Déposer un de leurs éléments, en allant le positionner à un endroit précis. Mais là, mon courage pourtant légendaire m'abandonne, et je renvoie les programmeurs concernés à des exemples de code traînant dans les bouquins ou sur le Net.

Bon, assez discuté, je sens que vous trépignez d'impatience à l'idée de mettre tout cela en pratique.

L'exercice **Lapins** est une introduction au Drag & Drop, qui ne pose aucune difficulté particulière.
Just a pawn in their game, en revanche, est nettement plus... stimulant.
Tout d'abord, il faudra absolument décompresser les deux fichiers *.ico livrés avec l'exécutable dans le même répertoire que celui-ci. Cette solution est certes inélégante, mais nous n'apprendrons que plus tard à procéder de la bonne façon.
Ensuite, le programme pose quelques petites difficultés, en premier lieu parvenir à susciter l'illusion que l'on promène les pions. Cela vous rappelle-t-il une remarque faite un peu plus haut ?

Partie 9

La chasse aux bugs

Programmer est une activité fort distrayante, mais chacun sait que ce n'est jamais que le meilleur moyen pour faire un élevage de bugs. Si on veut que l'application fonctionne, il reste, après l'avoir écrite - et en partie, tout en l'écrivant - le délicat problème d'exterminer méthodiquement les vilaines petites bêbêtes qui nous pourrissent l'existence. À cette fin, VB.NET propose un certain nombre d'ustensiles qu'il serait bien présomptueux de négliger.

Les bugs se répartissant en trois différentes espèces, et chacune ayant des moeurs différentes, les techniques de chasse, les appâts et les pièges devront être adaptées à chaque type de bestiole.

1. Les erreurs de syntaxe

Nous avons vu que l'éditeur de code de Visual Basic signalait en temps réel les erreurs de syntaxe (et plus généralement, les erreurs de compilation), et qui plus est, qu'il précisait la nature de ces erreurs, sous forme d'info-bulle. De ce côté, donc, pas de souci, voilà une affaire réglée : les bugs de syntaxe seront la première espèce à être repérée, puis exterminée par vos bons soins. Je ne m'appesantis donc pas sur ce point.

2. Les erreurs de logique

Là, c'est tout de suite une autre paire de manches. Rappelons en effet que l'erreur de logique, animal sournois s'il en est, ne provoque pas à proprement parler un plantage de l'application, ou tout au moins, pas forcément. Elle peut aussi se contenter de générer un résultat aberrant, ou tout simplement faux, ce qui la rend d'autant plus difficile à débusquer qu'elle peut passer inaperçue.

Ainsi, une erreur consistant à faire un tour de boucle de trop dans le balayage du tableau se repèrera immédiatement par le plantage du programme. Mais une erreur de calcul de 4,23 % dans le surcoût de l'échelon inférieur de la prime d'huile de l'ouvrier qualifié à l'échelon 543 modifié 42-B, il faut parfois être vigilant pour la repérer.

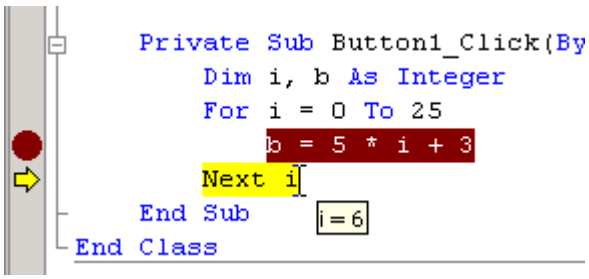
Remarque utile :
Avant toute chose, il faut vérifier que l'intégralité des outils de déboguage de VB a bien été mise en service, en allant dans le menu **Projet** - **Propriétés du projet** et en allant cocher la case **Générer les informations de débogage** si tel n'était pas le cas.

Toujours est-il que pour traquer toute cette engeance, VB propose plusieurs armes redoutables, pour la plupart disponibles notamment dans la barre d'outils **Déboguer**.

L'exécution pas à pas : pour commencer, la première question qu'il convient de se poser (et qu'on oublie trop souvent) : **le programme passe-t-il bien sur les lignes d'instructions que nous avons écrites, et le fait-il au bon moment ?** Un bon moyen de le savoir est de demander une exécution pas à pas. L'exécution ira alors de ligne en ligne, en n'avançant que lorsque nous lui en donnerons l'ordre formel, via la touche F8. Cette technique permet de repérer tout de suite les procédures dans lesquelles on ne rentre pas, et les mauvais branchements en général.

Le point d'arrêt : si on soupçonne un souci dans la procédure tout au fond en bas à droite derrière l'église, il peut être fastidieux de se fader dix minutes de pas à pas avant de parvenir à l'endroit suspect. Pour cela, une arme redoutable : le point d'arrêt, judicieusement posé d'un clic de souris dans la marge, juste avant l'endroit où les ennuis arrivent. On lance une exécution normale, qui s'interrompt au point d'arrêt. Et à partir de là, on peut y aller au pas à pas.

Un déboguage consiste souvent à pouvoir suivre au fur et à mesure de l'exécution les valeurs successives prises par une variable rétive. A cette fin, l'outil le plus simple et le plus puissant proposé par VB est tout bonnement... la souris ! En effet, en exécution pas à pas (et souvent, en ayant placé judicieusement un point d'arrêt), **il suffit d'amener la souris au-dessus d'une variable, ou d'une expression, pour qu'apparaisse dans un petit cadre jaune, la valeur de cette variable ou de cette expression**. Vous verrez, avec un peu d'habitude, c'est incroyable ce que ce petit truc peut rendre service :



3. Les erreurs d'exécution

Malgré tous nos efforts, et nos qualités de chasseurs, et même si notre code est optimisé aux petits oignons, il reste une espèce de bugs que nous ne serons peut-être pas parvenus à éradiquer : les **erreurs d'exécution**.

En effet, imaginons par exemple que notre application permette d'aller chercher un fichier quelque part sur la machine, avec une boîte de dialogue du genre du très classique **Fichier - Ouvrir**. Même en admettant que notre code soit parfait, nickel, eh bien s'il prend l'idée à l'utilisateur d'aller chercher un fichier sur la disquette alors qu'il n'y a pas de disquette dans le lecteur, ou sur le CD-ROM alors que le lecteur est vide, notre application se vautrera lamentablement en générant un message incompréhensible, mais néanmoins très effrayant.

Or, il faut toujours garder en tête les principes fondamentaux du développeur.

Principe n°1 :

Si une action de l'utilisateur est susceptible de faire planter une application, tout utilisateur accomplira précisément cette action dans les secondes qui suivent le lancement de l'application.

Principe n°2 :

Si le développeur est convaincu qu'aucune action de l'utilisateur n'est susceptible de faire planter son application, l'utilisateur se chargera de le détromper dans les minutes qui suivent le lancement de l'application.

D'où la nécessité absolue, lorsqu'on conçoit une application à vocation un peu sérieuse, d'utiliser l'arme ultime : le piège à erreur d'exécutions, c'est-à-dire le piège à **exceptions**.

Visual Basic .NET

Définition :

Une **exception** est une erreur d'exécution.

L'exception une **classe** de VB.Net, qui si une erreur d'exécution se produit, va générer un **objet** d'un type en rapport avec l'erreur.

L'idée de base, pour gérer ces erreurs, est en quelque sorte d'intercepter l'objet **Exception** qui sera créé en cas d'erreur, et de dire à VB ce qu'il doit faire dans ce cas. Cela s'effectue au travers d'un bloc **Try ... Catch ... End Try**, dont la structure la plus simple est la suivante :

```
Try
instructions susceptibles de provoquer une erreur
Catch variable as TypedException
instructions à exécuter en cas d'erreur
End Try
```

Remarque pertinente :

On peut écrire à la suite autant de blocs **Catch** que l'on souhaite.

Pour davantage de précisions sur ce point, voir quelques lignes plus bas.

On peut éventuellement ajouter à cette structure de base un bloc **Finally**, qui indiquera des instructions à exécuter en fin de procédure quoi qu'il se soit passé, qu'une erreur soit survenue ou non :

```
Try
instructions susceptibles de provoquer une erreur
Catch variable as Type_Exception
instructions à exécuter en cas d'erreur
Finally
instructions finales à exécuter dans tous les cas
End Try
```

Tout cela permettra donc au code de retomber sur ses pieds en cas de problèmes, et de faire qu'une erreur ne se traduise pas par un plantage, mais par une série d'instructions que vous aurez prévue à l'avance et qui permettra à l'application de poursuivre normalement sa route.

Parlons maintenant plus en détail des différents types d'exceptions.

Le plus général d'entre eux est le type **Exception**, qui les inclut toutes. Ainsi, une ligne comme :

```
Catch Erreur as Exception
```

...interceptera toutes les formes possibles et imaginables d'erreurs d'exécution. Mais on peut souhaiter à un traitement différencié selon la nature de l'erreur. S'agit-il d'un CD-ROM absent, d'un répertoire inexistant, d'un fichier absent ? On peut alors écrire autant de blocs **Catch** que de types d'erreurs que l'on souhaite différencier, sachant que ces types sont définis par Visual Basic (et en réalité, par la plateforme .Net). Voici un petit échantillon des types d'exceptions disponibles (on se réfèrera à l'aide ou à un bouquin pour une liste plus détaillée) :

Nom du Type	Nature de l'erreur

IO.FileNotFoundException	Fichier non trouvé
IO.DirectoryNotFoundException	Répertoire non trouvé
IndexOutOfRangeException	Indice en-dehors d'un tableau
ArithmeticException	Division par zéro ou dépassement de capacité

Bon, avec ça, vos applications n'ont plus aucune excuse pour se planter sur une erreur, quelle qu'elle soit.

Et vous non plus.

Visual Basic .NET

Partie 10

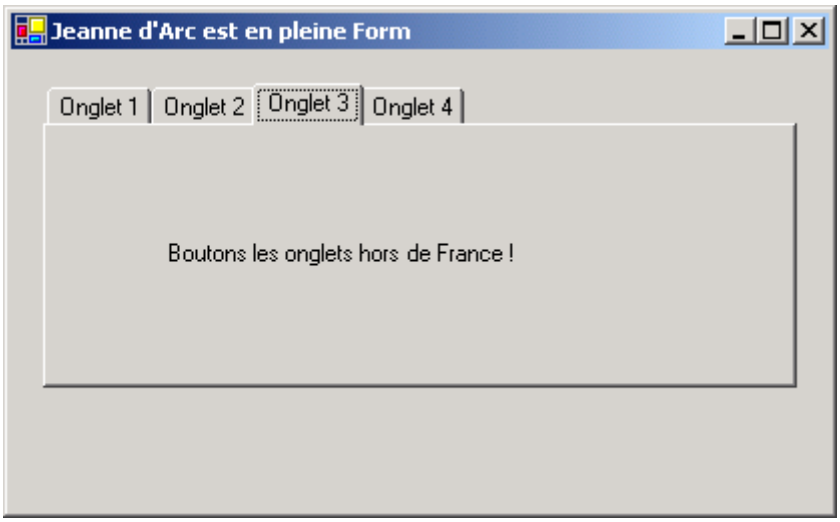
Toujours des contrôles

Continuons à fouiller la boîte à outils. Nous sommes encore loin d'avoir fait le tour de tous les couteaux à huit lames, des équerres courbes, des tire-bouchons à ressort et autres limes à épaissir proposées par VB. Voilà donc un chapitre bric-à-brac, dans lequel on va découvrir quelques ustensiles dont l'utilisation posera rarement de gros problèmes, et qui pourront très agréablement enrichir notre interface.

1. La classe TabControl

Il s'agit d'un contrôle qui va nous permettre de doter notre formulaire d'**onglets**, ce qui est bien connu pour améliorer tout autant l'ergonomie que le pot-au-feu.

Lorsqu'on le pose sur la Form, le **TabControl** prend l'aspect d'un rectangle qui délimite la partie de cette Form qui fonctionnera avec le système des onglets. Il est donc possible de laisser une zone (contenant des contrôles) qui sera visible quel que soit l'onglet sélectionné :



Pour ajouter des onglets en mode design, on peut passer soit par la propriété **Tabpages**, soit par **Ajouter un onglet**. Plusieurs propriétés, sur lesquelles je ne m'étendrai pas, règlent l'apparence de ces onglets. Au total, chaque onglet est représenté par un objet **Tabpage** et le nombre d'onglets est donné par la propriété **TabCount**. Le texte affiché sur chaque onglet est donné par sa propriété **Text**, avec la possibilité éventuelle de lui associer une image via un contrôle **ImageList**.

L'événement majeur pour un contrôle **TabControl** est **SelectedIndexChanged** qui indique bien évidemment que l'utilisateur a changé d'onglet. L'onglet actif est donné par la propriété **SelectedIndex**.

Bref, vraiment rien de méchant, et c'est encore un beau joujou supplémentaire.

2. Les classes HScrollBar, VScrollBar et TrackBar

Ces trois classes permettent à l'utilisateur de déterminer une valeur en maniant un curseur, qu'il s'agisse d'une barre de défilement horizontale (**HScrollBar**), verticale (**VScrollBar**), ou de cette barre de réglage un peu particulière (**Trackbar**) :

De ces contrôles très faciles à manipuler, il y a bien peu à dire :

- deux propriétés, **Minimum** et **Maximum**, déterminent la valeur associée à chaque extrémité.
- deux autres, **SmallChange** et **LargeChange**, règlent la valeur de l'incrément lorsqu'on déplace le curseur (selon qu'on clique sur la flèche qui se trouve au bout de la barre ou dans la barre elle-même)
- une propriété, **Value**, correspond à la valeur associée à la position actuelle du curseur.

A noter qu'il est possible d'afficher verticalement un contrôle **Trackbar**, en changeant sa propriété **Orientation**. Celui-ci possède de surcroît une propriété **TickStyle**, qui permet de modifier la trombine des graduations, et **TickFrequency**, qui règle leur intervalle.

L'événement privilégié associé à ces trois contrôles est le **ValueChanged**.

Et une fois qu'on a dit cela, on a fait le tour de la question.

3. La classe ProgressBar

Il s'agit de cette jolie chose censée faire patienter l'utilisateur - voire lui indiquer grosso modo où on en est - pendant une opération un peu trop longuette :

Là, c'est encore plus simple que précédemment, puisque l'utilisateur ne peut absolument pas agir sur ce contrôle. On retrouve donc les propriétés **Minimum**, **Maximum** et **Value**, mais guère plus, et pas d'événement particulier.

Circulez, y a rien, ou pas grand chose, à voir.

4. Les classes ToolTip et HelpProvider

Ces deux classes sont là pour donner une aide plus ou moins détaillée à l'utilisateur. Le **ToolTip**, c'est ce petit rectangle beige qui apparaît lorsqu'on stationne la souris pendant un certain temps au-dessus d'un contrôle :

En fait, il ne faut créer qu'un seul contrôle **ToolTip** par Form ; cet unique contrôle pourra servir ensuite pour l'ensemble des contrôles de cette Form. Les propriétés principales du contrôle tournent autour de la question du **Delay** : le temps au bout duquel le rectangle apparaît, le temps durant lequel il reste affiché, etc. Je ne détaille pas , parce que franchement, ça ne pose aucune difficulté. La seule astuce à connaître est le code qui permet d'associer un message à un contrôle donné. Il faudra pour cela écrire...

```
NomDuToolTip.SetToolTip(NomDuContrôle, Message)
```

Cette ligne devra naturellement être exécutée de préférence avant qu'il se passe quoi que ce soit d'autre, donc dans la procédure associée à l'événement **Form.Load**.

Quant au **Helpprovider**, que je ne cite ici que par acquit de conscience, il sert à "brancher" l'appui de la touche F1 d'un contrôle directement sur un fichier d'aide (*.chm ou *.htm), à la bonne page.

5. Les classes DomainUpDown et NumericUpDown

Ces deux contrôles ont un comportement très voisin de celui d'une liste qui aurait un look un peu particulier. Le premier, **DomainUpDown**, oblige l'utilisateur à choisir parmi une collection de **Strings**, et le second, **NumericUpDown**, parmi une série de nombres (simplement définis par un minimum et un maximum) :

Les propriétés et les méthodes de ces deux contrôles sont très voisines de celles des listes, pour ne pas dire qu'elles leur ressemblent comme deux couteaux (comme on dit en Alsace).

6. Les classes DateTimePicker et MonthCalendar

Il s'agit de deux contrôles spécialisés dans la saisie d'une date. Le premier des deux, **DateTimePicker**, est le plus polyvalent. Il se présente comme une liste déroulante déployant un calendrier :

Ce contrôle peut largement être paramétré :

fixation d'une date minimum et/ou maximum par les propriétés **MinDate** et **MaxDate**

empêchement de l'affichage du calendrier par **ShowUpDown**

changement de la valeur par défaut de la date, par **DateTime**

modification du format d'affichage de la date, par **Format**

modification de l'apparence du calendrier, par les six propriétés **Calendar**...

La date saisie est quant à elle stockée dans la propriété **Value**. Et le principal événement lié à ce contrôle est **ValueChanged**, provoqué par un changement de la valeur de la date.

Quant à **MonthCalendar**, c'est un **DateTimePicker** sans la zone de saisie, autrement dit un simple calendrier. L'avantage est que l'utilisateur pourra y sélectionner plusieurs dates (ce qui modifiera la valeur des propriétés **SelectionStart**, **SelectionEnd** et **SelectionRange**).

Bon, allez, on passe.

7. La classe Timer

Avec ce petit contrôle, on découvre un outil un peu déroutant au début, mais qui se révèle indispensable dans certaines situations :

En effet, jusque là, tous les événements que nous avons gérés étaient, d'une manière directe ou indirecte, provoqués par une action de l'utilisateur : un clic, une frappe au clavier, etc. Même lorsque l'événement était causé par l'exécution d'une ligne de code, celle-ci se trouvait dans une procédure elle-même déclenchée par une action de l'utilisateur (ou alors elle se trouvait dans une procédure elle-même appelée par une action de l'utilisateur, mais quelle que soit la longueur de la chaîne des causalités, au bout du bout du bout, il y avait un geste de l'utilisateur).

Or, dans certains cas, on peut souhaiter que **certains événements se déclenchent** (que certaines lignes de code soient exécutées) **indépendamment de toute action de l'utilisateur**.

Par exemple, imaginons que nous programmions un QCM en temps limité. Nous souhaitons que l'application se termine automatiquement au bout d'un délai donné, par exemple 30 minutes. Il va donc falloir, chaque minute, ou toutes les dix secondes, ou toutes les secondes, vérifier le temps écoulé depuis le début et en cas de besoin, prendre les mesures qui s'imposent en coupant le sifflet de l'utilisateur. Eh bien, c'est très exactement à cela que sert le **Timer**.

Il s'agit d'un contrôle capable de générer un événement à intervalle régulier, **événement qui sera donc totalement indépendant de ce que fait, ou ne fait pas, l'utilisateur**.

Ce contrôle est par définition invisible. A l'instar du **ToolTipText** ou de l'**ImageList**, Il ne se "pose" pas sur la Form, mais à côté.

La propriété **Interval** est celle qui règle l'intervalle de temps entre chaque événement généré par le **Timer**. Elle est exprimée en millisecondes. Et cet événement est tout simplement le **Tick**.

Bon, assez bavardé, amusons-nous un peu :

8. Les classes de boîtes de dialogue communes

Vous aurez sans doute remarqué que sous Windows, quel que soit le logiciel utilisé, ou presque, certaines commandes possèdent des airs de famille. Par exemple, lorsque vous faites **Fichier - Ouvrir**, **Fichier - Enregistrer**, **Fichier - Imprimer**, et quelques autres encore, non seulement les fonctionnalités proposées sont les mêmes, mais encore la présentation de la boîte de dialogue est rigoureusement identique d'une application à l'autre, au poil de nez près.

Cela ne peut signifier que deux choses. Soit les développeurs des différents logiciels prennent un soin maniaque à recopier leurs interfaces pour obtenir des copies conformes. Soit tous ces braves gens n'écrivent en fait rien du tout, et vont puiser du code, toujours le même, déjà programmé au sein de Windows. Eh bien devinez quoi ? C'est la deuxième proposition qui est la bonne.

De sorte que vous me voyez venir avec mes gros sabots : lorsque nous devrons insérer une de ces "boîtes de dialogues communes" dans notre application, nul besoin de la programmer nous-mêmes : Windows, via Visual Studio, nous offre la possibilité d'utiliser directement son code, au travers d'une série de classes. Les voici :

OpenFileDialog : correspond à la boîte de dialogue Fichier - Ouvrir

SaveFileDialog : correspond aux boîtes de dialogue Fichier - Enregistrer et Fichier - Enregistrer Sous

FolderBrowserDialog : correspond à un explorateur de répertoires (qui repère automatiquement la structure des répertoires de la machine sur laquelle tourne l'application)

FontDialog : correspond à la boîte de dialogue Police

ColorDialog : correspond à une boîte de dialogue donnant accès aux couleurs personnalisées

PrintDialog : correspond à la boîte de dialogue Fichier - Imprimer

PrintPreviewDialog : correspond à la boîte de dialogue Fichier - Aperçu avant impression

PrintPreviewControl : zone permettant d'afficher un Aperçu avant impression

PrintDocument : correspond à la boîte de dialogue Fichier - Imprimer

PageSetupDialog : correspond à la boîte de dialogue Fichier - Mise en page

Tous ces contrôles, hormis **PrintPreviewControl**, ne se positionnent pas sur la Form, mais en-dehors. Pour les déclencher, il suffit d'utiliser dans le code la méthode **ShowDialog**. Naturellement, il est possible de paramétrer ces boîtes de dialogue via leurs propriétés, et d'en récupérer les résultats, toujours via leurs propriétés. Je ne m'étend donc pas sur les détails techniques de leur utilisation, qui ne présentent pas grand intérêt pour la compréhension du langage, et qui ne posent aucune difficulté majeure pour leur mise en oeuvre... quand on a une bonne documentation.

Mais cela signifie encore une fois, je me permets d'insister, que toutes les tâches banales et courantes d'une application comme ouvrir, enregistrer, imprimer, et tutti quanti, se programment en un tournemain, vu que d'autres ont déjà fait le travail à votre place.

Partie 11

Les graphismes

Des graphismes avec VB.NET, il y aurait évidemment beaucoup à dire, et cela dépasserait largement le cadre de ce cours. Mais il y a tout de même quelques éléments de base pas si évidents, auxquels je voudrais consacrer ce chapitre... haut en couleurs, comme il se doit.

1. Couleurs et Propriétés

Même si jusqu'ici nous ne nous sommes pas particulièrement arrêtés sur cette question, votre proverbial sens de l'observation vous aura sans doute indiqué que la quasi-totalité des contrôles possèdent des propriétés désignant leur(s) couleur(s). Ainsi, **Backcolor** et **ForeColor** permettent de modifier respectivement la couleur d'arrière-plan et la couleur d'avant-plan (celle du texte) des contrôles.

Certes. Mais comment spécifie-t-on une couleur ? Il existe pour cela deux grands moyens. Lorsque nous réglons une des propriétés de couleurs à la main, dans la fenêtre design, nous voyons que s'offrent deux possibilités : soit le choix d'une couleur prédéfinie (via les onglets **web** et **system**), ou la composition d'une couleur entièrement paramétrable (via l'onglet **Personnaliser**, suivi d'un clic droit). Eh bien, le code nous propose globalement la même alternative.

1.1 La structure System.Drawing.Color

Elle correspond au choix d'une couleur au sein d'une palette prédéfinie (**web** ou **system**). Ces couleurs sont au nombre de 174, ce qui donne déjà un assez large choix. Les couleurs dites **web** porteront ainsi de jolis noms fleuris, tels **LightCoral**, **Orchid** ou **SeaShell**. Ces couleurs s'afficheront donc de la même manière sur toutes les machines où sera installée l'application. Les couleurs système, elles, porteront des noms rappelant leur utilisation dans Windows, comme **InactiveBorder**, **ScrollBar** ou **Menu**. Choisir une couleur **system** signifie que cette couleur variera d'une machine à l'autre, selon la manière dont l'utilisateur aura choisi de personnaliser Windows. Bon, jusque là, pas de problème.

En ce qui concerne le code, on retombe sur une technique déjà abordée. Soit on se tape l'intégrale, c'est-à-dire que pour mettre le fond de **Button1** en jaune, on devra écrire quelque chose du genre :

```
Button1.BackColor = System.Drawing.Color.LemonChiffon
```

Soit on opte pour la version abrégée, ce qui supposera d'avoir préalablement "importé" l'espace de noms par :

```
Imports System.Drawing
```

Ce qui permettra ensuite de se contenter d'un :

```
Button1.BackColor = Color.LemonChiffon
```

1.2 Définir une couleur personnalisée

Tout comme on peut le faire à la main, on peut dans le code définir sa propre couleur. Celle-ci doit respecter la structure des couleurs prédéfinies, à savoir être composée de quatre octets. La couleur peut donc être spécifiée par la méthode **FromArgb**, qui requiert en argument un ensemble de quatre nombres de 0 à 255 séparés par des points-virgules.

Le premier de ces arguments représente le degré d'opacité de la couleur (de 0 pour transparent, à 255 pour une couleur pleine). Les trois nombres suivants représentent dans l'ordre la quantité de rouge, de vert et de bleu. Ainsi, pour colorer un bouton en bleu, écrira-t-on :

```
Button1.BackColor = Color.FromArgb(255, 0, 0, 255)
```

Il est toutefois possible de passer la méthode **FromArgb** uniquement avec trois paramètres. Dans ce cas, ceux-ci désignent les couleurs, et par défaut, l'opacité est considérée comme maximale.

Et voilà de quoi s'amuser, bien qu'on soit censé avoir passé l'âge des coloriages.

2. Images et Contrôles

Là encore, il n'aura pas échappé à vos yeux de lynx qu'en mode design, un certain nombre de contrôles acceptent volontiers que leur fond soit constitué d'une image. C'est notamment le cas pour les Form et les Buttons, avec la propriété **BackgroundImage**.

Nous parlerons plus loin de la manière dont on peut affecter cette propriété par du code. En attendant, il faut dire quelques mots des contrôles dont le rôle spécifique est de contenir des images.

2.1 La classe ImageList (rappel)

Il y a tout d'abord l'**ImageList**, dont nous avons déjà fait la connaissance. Je rappelle que ce contrôle :

- est un véritable tableau d'images, qui peut donc en contenir autant qu'on le souhaite.
- reste invisible lors de l'exécution, ainsi que toutes les images qu'il contient.

Il ne peut donc servir que de "réservoir à images" pour les autres contrôles, qui iront y puiser les images nécessaires au fur et à mesure de leurs besoins, durant l'exécution de l'application.

2.2 La classe PictureBox

Si l'on veut qu'à un endroit de la Form, se trouve telle ou telle image, alors il faut utiliser le contrôle adéquat, à savoir **PictureBox**. Celui-ci possède une propriété **Image**, qui indiquera son contenu. Le contrôle **PictureBox** prend en charge les principaux formats d'image : JPEG, GIF, Bitmap, métafichiers (WMF), icônes... Il ne gère pas, en revanche, les vidéos.

Une propriété notable des **PictureBox** est **SizeMode**. Celle-ci peut prendre quatre valeurs, qui modifieront les propriétés du contrôle et/ou de l'image dans le cas où ceux-ci ne possèdent pas les mêmes dimensions :

aucun : ni le contrôle ni l'image ne changent de taille. Cela peut signifier que l'image sera rognée, et/ou que le contrôle débordera de l'image. L'image est alignée en haut à gauche du contrôle.

stretch : l'image est automatiquement étirée afin que sa taille s'adapte à celle du contrôle qui la contient.

autosize : la taille du contrôle est automatiquement adaptée à celle de l'image qu'il contient.

centerimage : les tailles de l'image et du contrôle ne sont pas modifiées, mais l'image est centrée par rapport au contrôle.

Remarque limpide :
Le contrôle **PictureBox** est incapable de gérer les images transparentes, quelle que soit la manière dont on s'y prend, et même si l'image qu'il contient est elle-même transparente.
Moralité, si l'on veut pouvoir gérer des images transparentes (et cela peut arriver plus souvent qu'on ne le croit), on sera obligé de faire appel à un autre contrôle que **PictureBox**.
Rendez-vous dans ce chapitre pour en savoir plus.

3. Gérer intelligemment les images

On en vient à présent à la manière de gérer les images dans une application. Il y a deux stratégies, possédant chacune leurs avantages et leurs inconvénients, qu'il vaut mieux connaître avant de faire des choix discutables et pénalisants.

Le premier mouvement, lorsqu'on veut utiliser des images, c'est évidemment de les intégrer aux contrôles (**PictureBox**, **ImageList**, **Form**, etc. en mode design, donc en tant que propriétés par défaut de ces contrôles. C'est la manière la plus simple de procéder, qui va avoir deux conséquences :

les fichiers images vont être considérés par vb comme **une partie intégrante de la Form**. A l'enregistrement de celle-ci, les informations qu'ils contiennent seront donc recopiées dans un fichier annexe du fichier *.vb, portant le même nom et l'extension *.resx

les fichiers image vont être **directement incorporés à l'exécutable**. L'avantage, c'est que celui-ci n'aura jamais de problème pour les trouver. Mais l'inconvénient, c'est que cette méthode peut alourdir considérablement l'exécutable en termes d'octets occupés, et peut finir par créer un monstre incapable de tourner sur certaines machines à la puissance limitée.

Voilà pourquoi cette technique, si elle a l'avantage de la facilité, est à réserver aux petites images, et qu'elle doit être proscrite dès qu'on a affaire à des images volumineuses et/ou trop nombreuses.

Pour celles-ci, comment faire ? Pour s'y retrouver, il va falloir en quelque sorte raisonner à rebrousse-poil, en partant du but à atteindre pour remonter vers les moyens à mettre en oeuvre. C'est parti :

Au final, que voulons-nous ? Que le(s) fichier(s) image(s) dont l'exécutable va avoir besoin soient stockés indépendamment de cet exécutable, afin de conserver à celui-ci la sveltesse qui fait toute son élégance.

Pour cela, il faut que l'exécutable contienne une ou plusieurs instructions donnant l'ordre de charger tel ou tel fichier image dans tel ou tel contrôle. Cette instruction est la méthode **FromFile**, qui appartient à la classe **Image**. On écrira ainsi :

```
PictureBox1.Image = Image.FromFile (nom du fichier)
```

C'est là qu'arrive un petit souci. Le nom du fichier, cela sous-entend de devoir préciser le répertoire dans lequel il se trouve. Dans le cas contraire, cette instruction serait incapable de fonctionner et provoquerait une erreur. Il faut donc absolument :

spécifier correctement, lors de l'emploi de la méthode **FromFile**, le chemin relatif où va se trouver le fichier image par rapport à l'exécutable qu'est notre application.

faire en sorte que lors de l'installation de cette application sur différentes machines, et ce, quel que soit le répertoire où s'effectuera l'installation, le fichier image se trouvera bel et bien dans le chemin relatif voulu.

Pour le moment, nous ne traiterons que du premier point. Le second, ce sera pour un peu plus tard, dans le chapitre 13 de ce cours.

Prenons un exemple. Admettons que nous choisissions de regrouper toutes les images dont notre application aura besoin au cours de son exécution, dans un sous-répertoire de l'exécutable, appelé "Pic".

Le chargement du fichier "**Charlot.jpg**" dans le contrôle **Portrait** devra donc dire ; "*va chercher le fichier Charlot.jpg qui se trouve dans le répertoire Pic, celui-ci étant un sous-répertoire de l'endroit où tu es en train de t'exécuter.*" Ceci peut s'effectuer de la manière suivante (j'ai découpé le code en plusieurs instructions plus plus de lisibilité, mais on pourrait bien sûr aller plus vite :

```
Actuel = Directory.GetCurentDirectory
  Fic = Actuel & "\Pic\Charlot.jpg"
Portrait.Image = Image.FromFile (Fic)
```

Et voilà. Je rappelle que cette méthode suppose de paramétrer l'installation pour que le sous-répertoire **Pic** soit automatiquement créé et que le fichier **Charlot.jpg** y soit déposé. Mais le gros avantage, c'est que l'exécutable reste d'une taille raisonnable.

Remarque généraliste :
Cette technique doit être mise en oeuvre, avec quelques aménagements, pour tous les fichiers annexes dont une application peut avoir besoin : fichiers de données, vidéos, sons, etc.

4. Deux mots sur les méthodes graphiques

Je ne m'étendrai pas sur cet aspect, qui ne peut intéresser que des programmeurs se lançant des un type d'applications un peu spécialisé. Mais autant savoir que VB.Net donne facilement accès aux classes et aux méthodes de Windows qui permettent de tracer directement des points et des formes à l'écran.

Le premier point à savoir est que l'on peut, par du code, tracer aussi bien des graphismes en mode Bitmap (c'est à dire en faisant du point par point) qu'en mode vectoriel (en raisonnant sur des formes).

Le second point est qu'on peut dessiner à l'écran (sur un contrôle), à l'imprimante (sur un objet **PrintDocument**) ou en mémoire (le résultat restant ensuite disponible pour n'importe quel usage, y compris être enregistré dans un fichier).

En mode Bitmap, on peut ainsi positionner des points, des rectangles, des ellipses, tracer des traits paramétrables avec un crayon ou un pinceau virtuel, faire des dégradés.

En mode vectoriel, il sera possible de tracer des lignes, des polygones, des cercles, des ellipses, des courbes de Bézier...

Bon, bref, j'avais dit que je ne serais pas long sur le sujet, eh bien je vais tenir parole.

Partie 12

Les menus

Une application informatique sans menus, c'est un peu comme un restaurant sans ordinateurs. Euh... Je m'égare. Bon, avec ce tout petit, petit chapitre, on va pouvoir reposer un peu nos neurones surmenés.

1. Le concepteur de menus

Si l'on veut un ensemble de menus dans sa Form, il faut créer un objet (un seul) de la classe **MainMenu**, classe disponible sous forme de contrôle dans la boîte à outils. Bien sûr, on peut aussi ajouter un menu par du code, mais étant donné le peu de cas où cette procédure s'avère nécessaire, j'en passerai pudiquement les détails sous silence.

Une fois l'objet **MainMenu** posé sur la Form, le concepteur de menus permet, en cliquant sur ce **MainMenu**, d'entrer à l'intérieur toute la série des commandes dont on souhaitera disposer. C'est extrêmement facile et ergonomique, il suffit de cliquer à côté, ou en-dessous, de l'élément adéquat, pour pouvoir entrer d'autres éléments. Ce faisant, on crée des objets de la classe **MenuItem**. Ainsi, chaque objet **MainMenu** comprend un certain nombre d'objets de la classe **MenuItem**, c'est-à-dire d'éléments de menu.

A signaler que les menus sont organisés les uns par rapport aux autres sous la forme d'un emboîtement de collections : les menus principaux (typiquement : Fichier, Édition, Affichage, etc.) sont les **MenuItems** membres de la collection de l'objet **MainMenu**. De même, les menus "secondaires" comme Ouvrir, Enregistrer, Enregistrer sous, etc. seront membres de la collection du menu Fichier. Et ainsi de suite.

1.1 Propriétés des menus

Pour introduire une barre de séparation entre deux menus placés verticalement l'un sous l'autre, il suffit de créer un élément possédant comme propriété **Text** un simple tiret (celui du signe de la soustraction).

Pour créer une touche de raccourci (celle qui permettra, conjointement à **Alt**, de déclencher la commande en question), il suffira de placer le signe **&** avant la lettre concernée dans le texte du menu.

la propriété **Checked**, si elle vaut **True**, permet de faire figurer devant le menu un signe indiquant son activation. Si la propriété **RadioChecked** vaut également **True**, ce signe est un cercle plein.

les propriétés **ShortCut** et **ShowShortCut** permettent respectivement d'attribuer une combinaison de touches du clavier à un menu, et de les afficher à côté du menu en question.

1.2 Les événements liés aux menus

Il s'agit pour l'essentiel de l'événement **Click**, lorsqu'on parle d'une commande ordinaire.

Dans le cas d'un menu qui n'est rien censé déclencher, hormis l'apparition sur sa droite d'un sous-menu, on pourra utiliser si besoin l'événement **PopUp**.

Enfin, le simple fait qu'un élément soit en surbrillance, suite au passage de la souris ou du clavier, déclenche un événement **Select**.

2. Menus contextuels

Il s'agit du menu qui apparaît lors d'un clic droit de la souris, et dont le contenu varie selon l'endroit où se trouve le curseur. Pour qu'un, ou plusieurs contrôles disposent d'un menu contextuel, la procédure à suivre est très simple.

On commence par poser sur la Form autant de contrôles **ContextMenu** qu'on aura besoin de menus contextuels différents.

On édite chacun de ces menus, en y créant les différents éléments voulus, via la même interface que pour les **MainMenu**.

On affecte à chaque contrôle le **ContextMenu** qui lui revient par sa propriété... **ContextMenu**.

Évidemment, tout cela peut également être réalisé par des liges de code, mais là encore, nous passerons tout cela sous silence, l'intérêt en étant réservé aux spécialistes.

A noter qu'un **ContextMenu** génère un événement **PopUp** lorsqu'il apparaît, et que tout comme pour un menu ordinaire, le clic sur l'un de ses éléments provoque un événement... **Click**.

Je vous l'avais bien dit, c'était un petit chapitre tranquille. Enfin, jusque là...

Soyons clairs, l'exercice qui suit est une vacherie.
Pas tant à cause de la structure des menus, qui se construit facilement grâce au concepteur dont nous venons de parler. Mais le "branchement" de chaque élément de menu ainsi créé, même s'il peut laborieusement être fait à la main, devra être ici entièrement programmé.
Ce qui pose le problème de l'exploration des collections de menus, et des sous-collections dans les collections, et des sous-sous-collections dans les collections... autrement dit de la mise en oeuvre d'un traitement **récursif** pour balayer l'arborescence des menus.
C'est un peu difficile quand on n'a pas l'habitude, mais il y a certaines situations où il faut savoir comment employer cette technique. La preuve !

Visual Basic .NET

Partie 13

Distribuer une application

Admettons. Votre application est terminée. Vous avez bien bossé, vous avez regardé partout, il n'y a plus un bug qui traîne, ce coup-ci, c'est sûr, c'est fini, votre calvaire s'achève. Eh bien, pas tout à fait. En effet, il va falloir s'assurer que nous pouvons produire un fichier, ou un ensemble de fichiers, qui s'installe correctement sur n'importe quelle machine (équipée de Windows), de sorte que le monde entier puisse profiter de votre oeuvre. Or, l'affaire n'est pas toujours évidente.

1. Compiler le projet

Pour compiler un projet, apparemment, rien de plus simple : il suffit de déclencher la commande **Générer le projet** pour disposer d'un joli exécutable en ordre de marche. Mais deux ou trois points méritent tout de même qu'on s'y arrête, car ils peuvent changer des détails... ou des choses plus importantes.

La commande **Propriétés du Projet** va notamment nous donner la possibilité de préciser quelques informations, dont :

le nom du fichier exécutable

l'icône qui représentera cet exécutable (en choisir une jolie plutôt que laisser l'immonde rectangle par défaut)

le type de compilation qui va être utilisé. Sans entrer dans les détails techniques, il est préférable de choisir l'option **Release** dans la **Configuration de solutions** : l'exécutable créé sera optimisé, donc moins encombrant.

2. Créer un programme d'installation

Tout cela est bien beau, mais pour peu que le projet soit un peu joufflu, cela risque fort de ne pas répondre à tous nos problèmes. Rappelons-nous ce que nous avons vu deux chapitres plus haut, à propos des images, des vidéos, des sons ou des fichiers de données (voire des fichiers de polices de caractères, et bien d'autres choses encore...) utilisés par notre application : nous ne pouvons - ou ne devons - pas les intégrer à l'exécutable, et nous devons nous assurer que sur toute machine où se fera l'installation, ils seront bien rangés dans le répertoire où le code ira les chercher.

Cela ne peut se faire qu'en créant non seulement un simple exécutable, mais aussi **un véritable programme d'installation**, comprenant également tous ces fichiers annexes et les informations sur l'emplacement où ils devront être installés. Fort heureusement, Visual Studio nous fournit gracieusement l'outil adéquat pour confectionner un joli paquet-cadeau.

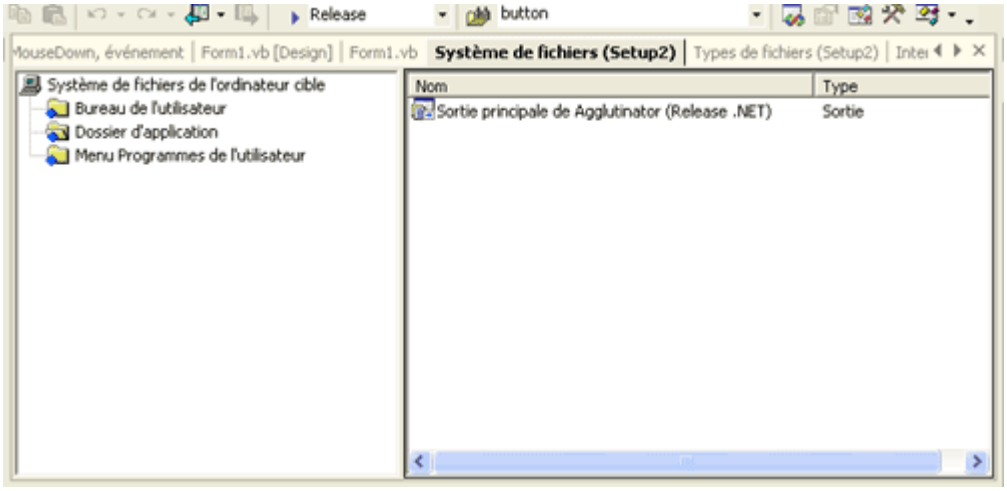
Remarque :

La première condition pour que votre exécutable tourne sur une machine, c'est que le Framework .Net y soit installé. Mais cela, ce n'est pas à vous de vous en préoccuper. En l'absence de ce Framework, le programme d'installation refusera de s'exécuter et enverra un message circonstancié à l'utilisateur afin qu'il le télécharge.

Reprenons.

Pour créer un programme d'installation, il faut **Ajouter un nouveau projet** au sein de la solution contenant le projet que vous voulez déployer. Mais ce projet sera d'un genre particulier : il faut préciser qu'il s'agit d'un **Projet de configuration** (on pourra, selon les cas, lui préférer éventuellement **l'assistant**, qui aboutit à des résultats à peu près comparables).

Ensuite, nous nous retrouvons illico dans la fenêtre **Système de fichiers**. Il s'agit d'un explorateur en deux volets, qui figure la machine de destination de notre application :



Dans le volet de droite, nous pouvons à présent poser les fichiers de notre choix, en sélectionnant au fur et à mesure dans le volet de gauche les répertoires de notre choix (on remarque qu'un des répertoires proposés est précisément le répertoire dans lequel sera installé l'exécutable - rien de plus facile, donc, que de créer des sous-répertoires et d'y mettre ce qu'on veut). Il y a également moyen de prévoir quelques opérations un peu particulières, comme poser un raccourci sur le bureau, voire associer certaines extensions avec notre application (mais là, ça commence à devenir un peu pointu pour le niveau où nous en sommes).

A noter que si dans le volet de gauche, trois répertoires de la machine-cible sont proposés par défaut, on peut en rajouter toute une série, qui couvrent les besoins les plus évidents.

Par ailleurs, les propriétés du projet de configuration permettent de préciser plusieurs choses, dont le **nom** qui apparaîtra publiquement comme celui de notre application.

Je signale pour mémoire la fenêtre **Interface utilisateur**, qui possède également son importance : c'est là que nous pouvons paramétrer en partie les boîtes de dialogue qui s'afficheront lors de l'installation, en ajoutant quelques commentaires, en modifiant l'ordre et le contenu des questions, etc. même si des modifications un peu importantes réclament un doigté et des connaissances que nous n'aborderons pas ici.

Au bout du bout du compte, lorsque tout est bien paramétré, qu'on est fin prêt, on peut y aller : il faut alors **compiler** (générer) **le projet de configuration d'installation lui-même**. On se retrouve alors avec différents fichiers, dont la plupart n'ont comme rôle que de contenir des informations de déboguage. Un seul compte pour nous : le **setup.msi**, c'est-à-dire l'installateur de notre beau projet.

Ben voilà, on a fait le tour de la question.

Partie 14

Les fichiers texte

Comme nous l'avons vu lors d'un excellentissime cours d'algorithmique, malgré les progrès et la sophistication croissante des langages de programmation, le fichier texte reste la voie la plus rustique - mais aussi une des plus sûres - pour stocker des données entre deux exécutions d'un programme. Aussi allons-nous traiter de l'accès aux fichiers, en nous cantonnant à la technique de base : le fichier texte en accès séquentiel. C'est ce qu'on appelle un chapitre - relativement - reposant.

1. Les instructions liées aux fichiers texte

Bon, là, quand on connaît la musique - c'est-à-dire le concept du fichier texte - il n'y a plus qu'à apprendre la syntaxe. Le plus difficile, c'est d'arriver à choisir, parmi la pelletée d'instructions que propose VB pour traiter des fichiers texte (et ça ne manque pas, ils sont vraiment fous !) une bonne petite technique qui soit simple et qui marche bien. Bande de petits veinards, ce boulot, c'est moi qui me le suis cogné, et voilà le résultat de mes savantes recherches.

Pour ouvrir un fichier texte, le plus facile est de passer par l'instruction **FileOpen**, selon la syntaxe suivante :

```
FileOpen(numéro, nom du fichier, mode)
```

- **numéro** est un entier compris entre 1 et 255 (ça devrait suffire assez largement...)
- **nom du fichier** est une chaîne de caractères
- **mode** est une valeur choisie parmi **OpenMode.Input** (ouverture en lecture), **OpenMode.Output** (ouverture en écriture) ou **OpenMode.Append** (ouverture en ajout)

Ensuite, pour recopier la ligne suivante du ficher numéro 3 dans la variable Toto, on écrira :

```
Toto = LineInput(3)
```

De même, pour recopier la variable Tutu en tant que ligne dans le fichier numéro 5, on écrira :

```
PrintLine(5, Toto)
```

La fonction retournant une valeur booléenne indiquant si l'on se trouve à la fin du fichier est évidemment **EOF()**, avec le numéro dudit fichier en paramètre. Enfin, pour fermer le fichier numéro 1, on emploiera :

```
FileClose(1)
```

Voilà. On ne peut pas dire que tout cela fasse bien mal à la tête.

2. Récupérer les lignes dans des variables

C'est là qu'on commence à rire pour de bon.

En effet, jusque là, dans tout langage civilisé, une des stratégies qui fonctionnait en toutes circonstances, pour le traitement des fichiers texte comme pour celui des fichiers binaires, consistait à travailler avec des structures. Ces structures étaient définies de manière à "coller" avec la manière dont les données étaient rangées dans les fichiers.

En ce qui concerne les fichiers texte, cela revenait à organiser les données - dans le fichier - sous la forme de champs de largeur fixe. Et à employer, en mémoire vive, une structure elle aussi "de largeur fixe", composée de chaînes de caractères au nombre de caractères déterminés une bonne fois pour toutes. Tout n'était peut-être pas pour le mieux dans le meilleur des mondes possibles, mais disons qu'on s'en approchait tranquillement. Eh bien, la nouvelle neuve du jour, c'est qu'avec VB.Net, il n'existe plus de chaînes de largeur fixe. Adieu donc les structures telles que nous les avons connues, et adieu les fichiers texte organisés sous forme de champs de largeur fixe. C'est beau, le progrès, non ?

Alors, à défaut de pouvoir faire ce qu'on veut, on n'a plus qu'à se résoudre à faire ce qu'on peut. C'est-à-dire à se rabattre sur les fichiers texte organisés sous forme de champs délimités.

Petit rappel :

Cela signifie que sur chaque ligne (enregistrement) de notre fichier, les différentes informations (les différents champs) seront séparés par un caractère séparateur (on le choisit généralement parmi la tabulation, la virgule, le point-virgule, et quelques autres).

Il va donc falloir, à chaque lecture de ligne du fichier, procéder à un découpage de cette ligne entre les différentes variables qui la composent. Mais comme nous avons de la chance dans notre malheur, nous allons pouvoir nous dispenser de nous taper une boucle lisant cette ligne caractère par caractère, afin d'y localiser les occurrences du caractère séparateur (chose qui était inéluctable dans les langages traditionnels). VB .Net propose en effet deux fonctions de découpage et de recomposition automatique des chaînes, qui vont grandement soulager notre peine. Voyons donc :

Visual Basic .NET

La fonction **Split** permet de constituer automatiquement un tableau de Strings à partir d'une chaîne de caractères dans laquelle figure un caractère de séparation. Ainsi, en exécutant le code suivant :

```
Dim S as String
Dim R() as String
S ="Ceci/est/un/message/codé"
R = Split(S, "/")
```

On récupère un tableau R de cinq cases, la première contenant "Ceci", la seconde "est", la troisième "un", etc. Et VB.Net autorisant des écritures beaucoup plus synthétiques, nous pouvons réécrire le code ci-dessus en deux coups de cuiller à pot :

```
Dim S as String = "Ceci/est/un/message/codé"
Dim R() as String = Split(S, "/")
```

Éventuellement, rien ne nous empêche ensuite de rebasculer les différents éléments de R dans une structure, ou dans un tableau de structures, si nous trouvons cela plus pratique (et il y a neuf chances sur dix pour que cela le soit effectivement). Simplement, je réinsiste, il ne s'agira pas de structures comportant des chaînes de longueur fixe.

Dans l'autre sens, nous trouvons en quelque sorte l'exact inverse de la fonction **Split**, en la personne de la fonction **Join**, puisque celle-ci constitue une chaîne unique à partir d'un tableau de Strings et d'un séparateur. Ainsi, si je repars du tableau R constitué un peu plus haut et que j'écris :

```
Dim Z as String
Z = Join(R, ";")
```

Z contiendra "Ceci;est;un;message;codé". Magique, non ? Évidemment, on aurait même pu se contenter de :

```
Dim Z as String = Join(R, ";")
```

Résumons-nous :

Split permet de récupérer dans un tableau les différents champs à partir d'une ligne d'un fichier texte organisée par un caractère séparateur

Join permet, à l'inverse, de constituer une ligne organisée par un caractère séparateur à partir de différents champs rangés dans un tableau.

On peut donc dégager le processus-type pour le traitement d'un fichier texte sous VB.Net (sachant que ce fichier devra être organisé via un caractère séparateur) :

1. déclaration d'une structure adéquate
2. déclaration d'un tableau de cette structure, non dimensionné
3. boucle de lecture du fichier, au sein de laquelle, pour chaque ligne :
4. on lit la ligne suivante
5. on la découpe par **Split** dans un tableau de Strings (de stockage temporaire)

6. on agrandit le tableau des structures
7. on recopie les informations du tableau de stockage temporaire vers le nouvel emplacement du tableau de structures
8. fin de la boucle

Et voilà. Ensuite, on procède à tous les traitements nécessaires (en mémoire vive, comme il se doit). Lorsque c'est terminé, on recopie le tableau de structures vers le fichier :

1. boucle de parcours du tableau de structure
2. pour chaque élément, on recopie les informations dans un tableau de Strings temporaire
3. on constitue par **Join** une chaîne à partir du tableau temporaire
4. on copie cette chaîne dans le fichier
5. fin de la boucle

Vous auriez préféré que j'écrive le code ? Ben voyons... Il faut tout de même que je vous laisse quelques occasions de vous dérouiller les neurones.

L'exercice qui suit, sans être vraiment méchant, n'est pas particulièrement facile. Il demande de gérer un fichier texte - vous le trouverez ci-dessous.
Il pose également (et surtout ?) quelques problèmes liés à la gestion d'une seconde Form, problèmes que nous avions évité jusque là, et dont je ne traite pas dans ce site, mais qu'il faut bien aborder un jour ou l'autre. C'est ici un cas assez facile, sur lequel vous trouverez des indications sur le site de Microsoft.
Quant au score obtenu lors de ce grand jeu, il s'agit tout bêtement d'un nombre aléatoire entre 1 et 1000.

Partie 15

VB.NET et Windows

Là, c'est vraiment parti pour le grand frisson. Oubliés les montagnes russes, le train fantôme et les disques de Frédéric François : pour se faire des vraies sensations, rien de tel que... Visual Basic.

1. Les outils qui ne sont pas dans la boîte

Jusqu'à maintenant, à part quelques digressions au sujet des structures du langage (parfois copieuses, je vous le concède), ce cours a consisté à baguenauder nonchalamment parmi les différents contrôles proposés dans la boîte à outils pour découvrir de quoi ils avaient l'air.

Cependant, toute la vie d'un programmeur ne se résume pas aux contrôles de la boîte à outils. Il peut arriver que notre imagination débridée (ou celle de notre client) exige des traitements dont ces contrôles se révèlent bien incapables.

Alors, dans ces cas-là, qu'est-ce qu'on fait ? Eh bien, il y a grosso modo deux cas de figure :

- le contrôle qui fait ce dont vous avez besoin existe... ailleurs que dans la boîte à outils.
- ce contrôle n'existant nulle part dans l'univers connu, il n'y a plus qu'à pallier la carence en vous mettant vous-mêmes au boulot.

Bien entendu, fainéants comme nous sommes, nous avons tout intérêt à explorer soigneusement la première possibilité avant de nous lancer dans la seconde. Première possibilité donc, le contrôle existe, ne reste plus qu'à le dégoter. Alors, ces contrôles qui existent ailleurs que dans la boîte à outils, où sont-ils ?

1.1 Les autres contrôles de Visual Studio

D'une part, **dissimulés dans VB lui-même**. Un certain nombre de contrôles (d'intérêt divers, il faut bien le dire) sont en effet là, tapis dans l'ombre, en attendant que vous veniez fouiller pour vous en servir. Pour les ajouter à la boîte à outils, c'est extrêmement simple : il suffit de passer par la commande **Outils - Ajouter/Supprimer des éléments de la boîte à outils**. Attention, il existe deux onglets : autant l'onglet **Framework.Net** ne vous offrira pas un choix énorme, autant les composants **COM** représentent tout de suite une série de nouvelles possibilités autrement plus étoffée.

Bien entendu, les contrôles récupérés par ce biais ne disposent d'aucune documentation : ce serait trop facile. Ce sera donc à vous de tâtonner, de deviner, bref, de jouer les Sherlock Holmes, pour trouver la clé de ces nouveaux outils et les utiliser au mieux.

 **Remarque fumante :**

Parmi les contrôles disponibles par ce biais, se trouvent dans le Framework .Net trois petits bijoux qui permettent, en deux coups de cuiller à pot, de disposer d'un explorateur d'unités logiques (sous forme de liste déroulante), d'un explorateur de répertoires (sous forme de **Treeview**) et d'un explorateur de Fichiers (sous forme de **ListBox**). Ces trois contrôles sont fort utiles pour construire des boîtes de dialogue de type **Fichier - Ouvrir** un peu personnalisées. Pour une description du code nécessaire, c'est tout en bas de cette page qu'il faut regarder.

1.2 Des contrôles dans le vaste monde de l'Internet

Bon, mais voilà, si comme il est toujours possible, le contrôle de notre coeur ne se trouve pas dans cette liste, où chercher ? C'est très simple : **sur le net**. Il existe, prêts à être téléchargés, des tonnes de contrôles qui ont été développés, payants, shareware ou... carrément en libre accès.

Un contrôle, **c'est un fichier de type *.ocx** : quand on effectue une recherche sur le net, c'est souvent une bonne idée que de taper cette extension plutôt que "contrôle" dans Google.

Si la recherche est fructueuse, vous récupèrerez parfois un fichier *.ocx tout seul. Parfois avec un exécutable qui l'installera sur votre machine. Et parfois aussi, accompagné d'un petit fichier d'explications sur

son utilisation, voire d'un projet servant d'exemple. Pour les contrôles gratuits, c'est en quelque sorte à la fortune du pot.

Une fois le fichier installé à un endroit quelconque de votre machine, il faut le faire apparaître dans la liste des contrôles disponibles. Pour cela, même manip que précédemment, onglet **Com**, puis bouton **Parcourir**, afin d'aller localiser le fichier. Le contrôle apparaît ensuite coché, et il suffit de faire **OK** pour qu'il fasse une entrée fracassante dans la boîte à outils. Et voilà le travail.

Avec ça, on peut voir venir pour 99 % des situations non prises en charge par les contrôles de la boîte à outils. Citons par exemple, pêle-mêle :

la diffusion d'un fichier mp3

la diffusion d'un fichier vidéo

la gestion d'un conteneur d'image gérant la transparence (rappelons-nous que le **PictureBox** en est incapable).

etc.

Remarque :

Face à un traitement un peu déroutant, on a toujours intérêt à prendre quelques minutes pour se demander si ce traitement ne ferait pas partie des fonctionnalités de Windows, auquel cas les chances de trouver un contrôle qui permette de les appréhender sont multipliées.
Un exemple parmi tant d'autres : la programmation d'un jeu de cartes, type belote ou bridge.
Avant de se lancer dans la numérisation des cartes et autres opérations fastidieuses de ce genre, on peut pertinemment remarquer que Windows intègre en standard plusieurs jeux de cartes. Ce qui signifie que les cartes utilisées par ces différents jeux existent dans une dll unique, à laquelle il est possible de faire appel pour programmer n'importe quel autre jeu (hormis le tarot, bien sûr...)

Mais, évidemment allez -vous dire, bande de grincheux que vous êtes, 99% des situations, c'est bien beau, mais et le 1 % qui reste ? Ben le 1 % qui reste, c'est que personne n'a fait le boulot... et que vous allez devoir vous y coller vous-mêmes.

En attendant, une petite récréation :

Pour cet exercice, plusieurs remarques s'imposent.
Tout d'abord, celui-ci faisant appel à un composant **ocx**, un simple exécutable ne suffit pas pour le faire fonctionner : le composant en question doit évidemment être lui aussi installé sur la machine de destination pour que le programme s'exécute convenablement. Voilà pourquoi le lien de l'exécutable pointe ici sur un véritable installateur, conçu avec la technique vue au chapitre 13.
Ensuite, le programme lui-même se divise en deux parties. La Tout d'abord, l'interface et les fonctionnalités, que vous devrez obligatoirement traiter. Puis ensuite, une conclusion plus purement algorithmique dédiée au calcul du résultat (paire, brelan, etc.). Vous pourrez laisser celle-ci de côté si vous avez peur de risquer une tendinite des neurones.

Exercice	Exécutable	Sources

Poker		

2. Quand il n'y a pas d'outils...

Lorsque vous êtes en pays étranger, que vous ne parlez pas un mot de la langue locale, et que vous voulez demander à un autochtone d'accomplir une tâche quelconque, vous avez trois solutions :

vous trouvez un interprète

vous prenez votre dictionnaire français - syldavien, et vous vous débrouillez pour faire des phrases sans interprète.

Eh bien, avec VB .Net, c'est un peu la même chose... sauf qu'il n'y a pas de dictionnaire ! Ou tout au moins, pas de dictionnaire fiable et complet. Reprenons.

La langue étrangère inconnue parlée par les indigènes, c'est évidemment Windows, et ses centaines de programmes compilés : les dll. Ces programmes sont utilisés par Windows lui-même, mais ils peuvent également l'être par n'importe quel logiciel qui le souhaite. Encore faut-il savoir comment on s'adresse à une dll. L'interprète, qui permet de dialoguer avec les dll pour leur faire faire ce qu'on souhaite, c'est évidemment le **contrôle**. Dès lors :

soit ce contrôle existe déjà (ce sont les différents cas dont nous avons parlé jusque là)

soit le contrôle n'existe pas, alors nous pouvons éventuellement le fabriquer

La fabrication de contrôles est une tâche certes exaltante, mais pas toujours facile. En fait, il s'agit d'un cas particulier du problème plus général de la fabrication de nos propres classes, problème sur lequel nous allons nous pencher... dans le chapitre suivant.

Dans celui-ci, je vais simplement dire quelques mots de la dernière possibilité, celle qui consiste à appeler directement les dll de Windows depuis un programme VB, sans passer par l'intermédiaire d'un contrôle. Une telle technique est toujours assez laborieuse à mettre en oeuvre : les dll de Windows sont loin d'être toujours bien documentées, et il faut parfois déployer des trésors de patience pour dégoter, dans un obscur forum ou dans une revue confidentielle, la syntaxe qui vous permettra de faire tourner la dll voulue. C'est d'ailleurs pour cette raison que les contrôles existent : afin de présenter au programmeur une interface plus sympathique pour s'adresser aux dll de Windows.

Admettons donc que nous ayons besoin d'appeler la dll CHMOLDU, pour laquelle nos recherches désespérées n'ont pu nous donner aucun contrôle disponible.

Visual Basic .NET

Partie 16

Visual Basic .NET

Plus loin avec les objets

1. Retour sur l'analyse orientée objet

1.1 Quelques généralités

Jusqu'à maintenant, même si cela malmène notre amour-propre, il faut bien l'avouer : nous n'avons été que des petits joueurs. Nous nous sommes certes dépatouillés de bien des situations difficiles avec VB.Net, à commencer par la première d'entre elles : se repérer dans toutes ces notions de la programmation objet que sont les classes, les propriétés, les méthodes et les événements. Mais malgré les heures d'intense émotion que nous ont valu ces épreuves, il faut bien le dire, nous sommes restés au seuil de la véritable et ultime révélation.

En effet, jusqu'à présent, nous n'avons fait que manipuler des classes qui avaient été déjà créées par les développeurs de VB, ou à la grande rigueur par d'autres (celles que nous sommes allés pêcher sur le net). Ce n'est déjà pas si mal, me direz-vous. Certes, répondrai-je. Mais le véritable programmeur objet, c'est celui qui est capable non seulement de faire de l'assemblage, mais aussi de la conception ; qui est capable non seulement d'utiliser les classes déjà programmées par d'autres que lui, mais aussi de concevoir et de réaliser ses propres classes.

Maîtriser ce savoir-faire, cela suppose en réalité deux types de compétences distinctes, même si une seule personne peut tout à fait cumuler les deux.

- il faut être capable, face à un problème donné (une application à réaliser) de formaliser ce problème, d'imaginer quelles classes seront nécessaires, quelles seront les caractéristiques de ces différentes classes (leurs propriétés, leurs méthodes, les événements qu'elles devront gérer), l'articulation éventuelle qu'elles auront entre elles, bref : il faut produire une **analyse orientée objet** du problème.
- et il faut savoir traduire ces spécifications dans un langage (ici, le VB.Net) afin de réaliser concrètement les classes en question, pour qu'elles soient utilisables dans une ou plusieurs applications. Il s'agit là d'un savoir **purement technique**.

Des deux aspects, vous vous doutez que c'est le second, même s'il apparaît comme le plus rebutant, qui est en réalité le plus facile à maîtriser. Et c'est donc celui-là que je vais développer dans les lignes qui suivent. Car pour acquérir de bons réflexes et une bonne intuition sur l'analyse orientée objet, il faut du temps et de la pratique sur des cas concrets, toutes choses bien difficiles à transmettre via un site web, fut-il aussi extraordinaire que celui-ci. Cela ne nous empêchera pas de partir d'un exemple précis, ne serait-ce que pour que l'exposé technique qui va suivre ne soit pas totalement désincarné. Et puis, l'analyse objet, c'est comme le reste, il faut bien commencer par en faire si on veut pouvoir s'améliorer.

Pour la suite, nous allons donc partir d'un exemple que tout le monde connaît : **le jeu du Monopoly** (qu'on s'entende bien, il ne s'agit pas de le programmer intégralement, mais simplement d'y puiser nos petits bouts d'exemples).

1.2 Le Monopoly : une brève analyse procédurale

Pour programmer un Monopoly, on pourrait tout à fait adopter une démarche traditionnelle, c'est-à-dire **procédurale**. Dans ce cas, il faudrait s'interroger sur les informations à manipuler, et sur le type de codage le plus approprié. Pour figurer le plateau de jeu, on sent que le mieux serait que l'affaire tourne autour d'un tableau à une dimension, où quand on arrive à la case du bout (la rue de la Paix), on repart à la case zéro. On pourrait ainsi avoir par exemple un tableau de chaînes de caractères qui stocke uniquement la position des pions (codés A pour le premier joueur, B pour le deuxième, etc. et "" s'il n'y a pas de pions sur la case ; s'il y a plusieurs pions sur la même case, une simple concaténation "AC" pourrait facilement l'indiquer). A ce tableau codant le plateau de jeu pourrait également correspondre un autre tableau codant les noms des différentes cases, un troisième codant leur couleur, un quatrième codant le prix de la construction d'une maison, un autre codant le prix du loyer sur terrain nu, etc.

Pour coder la totalité des informations nécessaires, **on aurait ainsi, à vue de nez, une dizaine de tableaux fonctionnant un quelque sorte en parallèle** : lorsque le joueur tombe sur la case numéro i du plateau de jeu, on sait que les informations utiles se trouvent dans les cases numéro i des différents tableaux. C'est une manière de faire, peut-être pas la plus élégante, mais qui permettra sans doute de parvenir à un résultat qui fonctionne.

Une alternative à cette démarche, toujours en restant dans le cadre de la programmation procédurale, serait de nous donner, plutôt que plusieurs tableaux côte à côte, **un seul tableau, où chaque case regrouperait l'ensemble des informations correspondant à ladite case sur le plateau de jeu**. Autrement dit, où chaque case ne serait plus une information de type simple, mais un ensemble d'informations de type simple... c'est-à-dire **une structure**. Et l'ensemble du plateau de jeu serait donc un tableau de structures.

Nous aurions donc une structure (la case) regroupant toutes les informations nécessaires : une chaîne de caractères pour coder les pions présents, un numérique ou un caractère pour en coder la couleur, une chaîne pour en coder le nom, un numérique pour en coder le prix d'achat, un autre pour le prix de la maison, etc. Et notre plateau de jeu serait un tableau (à une dimension) de cette structure.

Cette deuxième solution, tout en restant dans les limites de la programmation procédurale, est déjà beaucoup plus proche, dans son esprit, de l'approche objet que ne l'était la première (celle avec les différents tableaux). Cela ne doit pas nous étonner ! Rappelons-nous de la manière dont nous avions défini les classes (ou les objets) au tout début de ce cours : **une classe, c'est une structure plus du code**.

En regroupant nos informations en structures, nous avons donc fait la moitié du chemin qui nous mène à l'approche objet. Reste la deuxième moitié, celle du code.

1.3 Le Monopoly : une brève analyse objet

Partons de notre structure, c'est-à-dire de notre case du plateau de Monopoly.

En définissant l'ensemble des informations devant être stockées pour chacune des cases, nous avons donné les éléments de cette structure. Cependant, si nous considérons dorénavant chaque case non plus comme une structure, mais **comme un objet** formé à partir de la classe "case", alors nous pouvons tout aussi bien dire que nous venons de définir les **propriétés** de cette classe.

Normalement, si vous avez digéré tout ce que nous avons fait jusque là, cette affirmation ne doit vous poser aucun problème. Si c'est le cas, vous n'avez plus qu'à tout relire depuis le début (mais non, je plaisante ! Enfin... allez savoir).

La nouveauté, maintenant que chaque case est un objet, c'est que nous allons pouvoir réfléchir à toutes les actions du jeu, à tout ce qui peut se passer, et incorporer ces actions, sous forme de code, dans la classe case. Pour cela, il va nous falloir distinguer :

les méthodes : ce sont les actions de jeu **qui modifient uniquement les caractéristiques de la case concernée**. Le code correspondant pourra être donc intégré à la classe case, donc aux objets qui seront créés d'après cette classe.

les événements : ce sont les actions de jeu **qui auront des répercussions sur d'autres cases, ou sur la banque, ou sur les autres joueurs, etc.** Dans ce cas, on ne pourra bien sûr pas intégrer le code dans la classe, mais ils devra figurer en dehors, dans l'application proprement dite.

Cette distinction entre méthodes et événements, elle aussi, ne doit normalement pas vous poser de problèmes particuliers, si vous réfléchissez bien à tout ce que nous avons déjà fait avec des classes déjà préfabriquées. Les méthodes des classes, pré-écrites dans les classes elles-mêmes, étaient toujours un moyen de modifier l'objet concerné, et uniquement lui. Les événements constituaient quant à eux une "porte ouverte", nous permettant de taper le code de notre choix, qui pouvait du coup concerner n'importe quel autre objet que celui ayant déclenché l'événement.

Cela dit, quand on regarde de plus près, la distinction n'est pas toujours évidente. Ainsi par exemple, prenons deux actions du jeu et tâchons de savoir si nous devons en faire des méthodes ou des événements :

acheter un terrain : il faudra noter que le terrain est dorénavant attribué au joueur Duchemol (si nous en faisons une méthode, celle-ci devra donc exiger le nom, ou son numéro en argument). Problème, le compte en banque du joueur devra également être délesté de la somme adéquate. A partir de là, une chose est sûre : nous ne pouvons nous contenter d'une méthode. Pour le reste, cela nous laisse deux possibilités. La première est de n'en faire qu'un événement, lequel gèrera les deux aspects (modification des caractéristiques de la case et du compte en banque de l'acheteur). La seconde consiste à décomposer l'affaire en deux parties : une méthode pour gérer ce qu'il advient du terrain, et un événement pour gérer la modification du compte en banque. En l'occurrence, les deux choix sont possibles, et ont chacun leur pertinence.

arrivée d'un pion sur la case : ceci peut en revanche être traité dans une simple méthode, puisqu'il suffira de modifier pour la case concernée la propriété adéquate. Cela dit, l'arrivée d'un

pion sur une case pouvant donner lieu à toute une série de conséquences (paiement éventuel d'un loyer, achat éventuel, etc.) on peut également choisir, comme précédemment, d'en faire un événement.

Comme on le voit, la frontière entre méthodes et événements est beaucoup plus floue que celle qui sépare les propriétés d'une part, les méthodes et les événements de l'autre. Bien souvent, on est amené à faire des choix sinon arbitraires, du moins stylistiques. Et bien souvent, en ce qui concerne au moins les classes fournies par Visual Studio, nous avons pu constater que les programmeurs n'avaient pas reculé devant la redondance, nous proposant pour une même notion à la fois une propriété, une méthode pour la gérer et un événement déclenché par sa modification (pensez à la sélection d'un item dans une liste). Mais abondance de biens nuit rarement, et après tout, une manière de voir les choses est de se dire que plus on a prévu de "branchements" lors de la création d'une classe, plus celle-ci sera facile à employer en toute circonstance.

2. Le code

2.1 Déclarer une classe

Il y a pour cela plusieurs possibilités, mais nous opterons pour la plus générale : celle d'une classe créée pour être utilisée par les différentes Form d'un projet donné, par exemple le Monopoly. Nous commencerons donc par créer une nouvelle classe, en utilisant la commande **Projet - Ajouter une classe**.

Nous constatons qu'apparaît alors un nouveau fichier, d'extension ***.vb**, portant le nom de la classe. Ce fichier possède toutes les caractéristiques des fichiers Form, dont nous sommes familiers, à cette différence que pour sa part, il ne contient que du code (on parle parfois à ce propos de **module de classe**).

Le code de chaque classe que nous créerons sera ainsi stocké dans un fichier particulier. Lequel aura pour particularité d'être encadré par les deux instructions suivantes :

```
Public Class NomdelaClasse

End Class
```

Le mot clé **Public** signifiant ici que cette classe sera accessible depuis n'importe quelle Form (ou module) du projet.

A partir du moment où une classe est créée, nous pouvons **instancier** cette classe dans notre code, c'est-à-dire créer des objets à partir de cette classe, en tapant, comme nous l'avons toujours fait :

```
Dim Toto As New NomdelaClasse
```

Nous vérifions au passage que **NomdelaClasse** apparaît dorénavant dans la liste déroulante de l'éditeur de code après le mot **New**, preuve que la classe est bien reconnue et disponible. Comme quoi, on n' a pas bossé pour rien.

2. Les propriétés

Une propriété est tout simplement une variable de la classe qui a été déclarée comme publique. Une classe gère donc deux sortes de variables : les privées, celles qui lui servent pour ses propres opérations, et qui sont invisibles depuis l'extérieur (depuis le programme qui utilise cette classe), et les variables publiques, qui sont donc les propriétés.

Il suffit donc de taper dans le code de la classe une ligne comme :

```
Public Pions as String
Public Prix as Long
```

...pour que **Pions** et **Prix** soient considérés comme les deux propriétés de cette classe, et que je puisse entrer dans ma Form, à la suite de la ligne ayant créé **Toto** :

```
Toto.Pions = ""
Toto.Prix = 12000
```

Et là encore, je vérifie que sitôt entré le point qui suit **Toto**, s'affiche une superbe liste déroulante qui comporte les deux propriétés **Pions** et **Prix**. Elle est pas belle, la vie ?

Bon, là, je n'ai fait qu'employer le minimum du minimum (ce qui s'avère toutefois suffisant dans bien des cas). Mais parfois, on peut souhaiter vouloir introduire un certain nombre de limitations ou de vérifications.

Une première solution consistera à avoir recours à une **énumération**. Nous n'avons jusque là guère utilisé cet outil, mais il est temps de découvrir, ou plutôt de redécouvrir, son existence. Redécouvrir ? En effet. Rappelez-vous de ce que j'avais dit sur les constantes VB : ce sont des mots-clés, associés par le langage à des entiers. Eh bien, en réalité, ces constantes VB sont **les membres d'une énumération**. Voyez plutôt.

Pour nos cases de Monopoly, la couleur ne peut prendre qu'un certain nombre de valeurs bien définies. Mettons, pour faire simple, bleu marine, bleu clair, violet, orange et rouge. Cela signifie que la propriété couleur, que je vais créer dans un instant, n'est censée accepter que l'une de ces cinq valeurs, et rien d'autre.

Commençons par créer cette énumération, dans la classe. Avant même les déclarations de variables (de propriétés), tapons :

```
Public Enum MesCouleurs as Integer
    bleu marine = 0
    bleu clair = 1
      violet = 2
      orange = 3
       rouge = 4
    End Enum
```

Nous venons de créer un nouveau type, qui associe un texte à cinq nombres, et qui ne peut admettre comme valeur que l'un de ces cinq textes, ou l'un de ces cinq nombres. Exactement comme pour les **vbOK** et autres **vbYesNoCancel** que nous avons longuement fréquentés.

Il ne reste plus qu'à créer la propriété **Couleur**, en nous basant sur ce type **MesCouleurs** :

```
Public Couleur As MesCouleurs
```

A partir de là, dans le code de la Form, lorsque je tape une ligne comme :

```
Toto.Couleur = ...
```

Je vois apparaître, après le signe égal, une liste déroulante comportant les cinq valeurs autorisés par mon énumération **MesCouleurs**. Magique, non ?

Pour terminer sur ce point, il existe des moyens encore plus puissants d'introduire des vérifications et des traitements divers au niveau des propriétés. Pour cela, il est indispensable de sortir l'artillerie lourde, à savoir de ne plus déclarer la propriété comme une simple variable, mais **comme une procédure Property**. Et c'est dans cette procédure qu'on pourra introduire le code voulu.

Pour mémoire car je m'arrêterai là, le bloc d'instructions associé à la lecture d'une propriété s'intitulera **Get ... End Get**, et celui associé à son écriture sera **Set ... End Set**. Mais pour une description plus détaillée du maniement de ces choses-là, je vous renvoie sur l'aide, ou sur des manuels plus complets que ce malheureux site.

3. Les méthodes

Si les propriétés ne sont finalement que les variables publiques d'une classe, **les méthodes ne sont, elles, rien d'autre que les procédures et les fonctions publiques de cette classe**. La boucle est bouclée, et comme le disait le sage, tout est dans tout et réciproquement.

Voyons par exemple la méthode permettant d'ajouter un nouveau pion à une case. Nous passerons en paramètre de cette méthode le nom du joueur ("A", "B", "C", etc.) et la procédure s'occupera d'ajouter la lettre correspondante à la propriété Pions. Écrivons le code dans notre classe :

```
Public Sub PionArrive(ByVal P as String)
    Pions = Pions & P
End Sub
```

De là, l'arrivée d'un pion dans la case **Toto** pourra aisément être programmée dans la Form par la ligne :

```
Toto.PionArrive ("A")
```

Et voilà le travail. Et comme d'hab, dès qu'on tape le point qui suit **Toto**, on voit apparaître **PionArrive** dans les choix possibles. Et dès qu'on ouvre la parenthèse, on nous signale qu'il faut entrer un paramètre de type **String**. Ça a beau être la troisième fois que ça nous arrive en peu de temps, il n'y a pas à dire, ça fait encore quelque chose.

Les mauvais esprits, ceux qui d'habitude roupillent dans le fond près du radiateur mais qui viennent de se réveiller juste pour dénigrer mes propos, feront sans doute remarquer que c'était bien la peine de créer une méthode exprès, alors qu'un simple :

```
Toto.Pions = Toto.Pions & "A"
```

...écrit dans la Form aurait eu exactement le même résultat. A ceux-là, je répondrai tout d'abord qu'il faut bien commencer par des choses simples, voire un peu niaises, avant de passer à celles qui sont plus compliquées. Ensuite, que les classes de Visual Studio sont truffées de méthodes redondantes par rapport au maniement immédiat des propriétés. Enfin, que c'est moi le chef, et que c'est moi qui décide ce qui est intelligent ou non.

Il suffit d'ailleurs de se poser le problème inverse pour voir l'intérêt d'une méthode : enlever un pion d'une case n'est pas aussi facile que d'en ajouter un (car s'il y a plusieurs pions sur la même case, donc plusieurs lettres dans la chaîne **Pions**, on ne sait pas en quel position se trouve celui qu'on doit supprimer). Dans ce cas, la méthode pourrait être écrite comme suit :

```
Public Sub EnleverPion(ByVal P As String)
        Dim x As String
        Dim i As Integer
            x = ""
    For i = 1 To Pions.Length
    If Mid(Pions, i, 1) <> P Then
        x = x & Mid(Pions, i, 1)
            End If
        Next i
    Pions = x
End Sub
```

De sorte que l'enlèvement du pion "C", dans la Form, s'écrira :

```
Toto.EnleverPion ("C")
```

Même les mauvais esprits commenceront à comprendre que **plus le code de la classe est complet et comprend de traitements, plus le code de la Form en sera clair, allégé et lisible**. En fait, on peut dire que si le découpage d'un traitement en sous-procédures et fonctions représentait un pas vers l'allègement et la rationalisation du code, sa structuration en classes, avec leurs propriétés et leurs méthodes, représente de ce point de vue une avancée supplémentaire. Et je conclurai cette envolée par l'aphorisme suivant, que je viens d'inventer et dont je ne suis pas peu fier : **la programmation objet, c'est le règne de la sous-traitance, où le donneur d'ordre (les procédures) délèguent au maximum aux classes tant le stockage des données que leur traitement**.

4. Les événements

Là, l'affaire se corse un tantinet. Mais on peut tout de même faire au mieux pour rester dans les limites du raisonnable.

Pour qu'un objet puisse générer un événement, il faut tout d'abord que cet événement, tout comme les propriétés et les méthodes, soit déclaré dans la classe. Ceci s'effectue par le mot-clé **Event**, suivi des paramètres nécessaires.

Par exemple, admettons qu'outre une méthode **AcheterRue**, que nous avons créée pour signifier par exemple que la rue **Toto** vient d'être attribuée au joueur B, nous souhaitions créer un événement

DébitCompte afin d'aller débiter le montant du compte en banque du joueur concerné du prix de la rue qu'il vient d'acquérir.

Cet événement étant généré par une méthode de la classe **CasesMonopoly** (c'est l'exécution de la méthode **AcheterRue** qui doit provoquer l'événement **DébitCompte**, il convient de déclarer l'événement dans la classe **CasesMonopoly**. Les paramètres nécessaires pour effectuer le traitement étant le nom du joueur et le montant de la transaction, ceci s'effectuera ainsi :

```
Event DebitCompte (ByVal Joueur as String, ByVal Montant as Long)
```

Et c'est tout. Cependant, il va falloir à présent déclencher l'événement lorsqu'un achat est effectué. En admettant que l'appartenance d'une rue est stockée dans une propriété appelée **Possesseur** et que son prix d'achat le soit dans une propriété appelée **Prix**, la méthode **AcheterRue** ressemblera donc à quelque chose du genre :

```
Public Sub AcheterRue (J as String)
            Possesseur = J
        RaiseEvent DebitCompte(J, Prix)
            End Sub
```

Il ne reste plus, pour finir, qu'à écrire la procédure **DebitCompte** proprement dite. Où ça ? Eh bien, vu qu'il s'agit d'une procédure qui ne se rattache à aucune classe particulière, dans la Form, ou mieux, sur un **module** (un module est un emplacement destiné à stocker du code, mais qui, à la différence de la Form, ne comporte aucun aspect visuel ; si l'on veut, il stocke des procédures et des fonctions, mais pas de contrôles).

On écrira donc, en supposant que les comptes en banque des joueurs sont stockés sous la forme d'un tableau **Comptes()** de numériques, la case 0 correspondant au joueur A, la 1 au joueur B, etc. :

```
Public Sub DebitCompte(ByVal X as String, ByVal Y as Long)
                Dim Num as Integer
            num = Instr("ABCDEF", X) - 1
            Comptes(num) = Comptes(num) - Y
                End Sub
```

Ainsi, si l'on considère le trajet effectué par le nom du joueur, celui-ci aura finalement accompli un joli périple. En effet, le nom du joueur a été transmis depuis la procédure située dans la Form jusqu'à la méthode **AcheterRue** de l'objet **Toto**. Là, rejoint par le montant de l'achat, il a été transmis via l'événement **DebitCompte**, à la procédure correspondante, et traité à cet endroit... ouf !

Cela dit, l'avantage de cette architecture, c'est de découper au maximum le code, et de procéder par emboîtage successifs d'éléments (dont les classes) pouvant être testés et débogués au fur et à mesure de la construction.

On remarquera évidemment que dans le Monopoly, les cases ne sont pas du tout les seuls éléments à pouvoir être traités comme des objets. On aurait très bien pu faire de même avec les joueurs : auquel cas le compte en banque et le nom auraient été des propriétés de cette classe **Joueurs**, et auquel cas il aurait également fallu écrire le code précédent d'une manière assez différente. Mais bon, hein, le but de la manoeuvre, c'était de donner les clés. Après, c'est à vous d'apprendre à ouvrir de plus en plus de portes (tiens, c'est beau, ça, comme image, je la garde).

3. Créer ses propres contrôles

Nous pouvons maintenant ajouter quelques mots au sujet d'un problème que nous avions laissé de côté au chapitre précédent : créer nos propres contrôles.

Les contrôles n'étant jamais qu'un type particulier d'objets, tout ce que nous venons de voir à l'instant à propos des classes reste évidemment valide à propos de contrôles. Cependant, les contrôles présentant certaines particularités, leur création peut s'avérer selon les cas plus facile, ou plus difficile, que la création des classes.

Plus facile, car on peut partir d'un contrôle existant, on se contente d'apporter quelques modifications. Plus difficile, car le contrôle est le plus souvent une classe dont les objets doivent avoir une existence à l'écran, et réagir à une pléthore d'événements.

On peut ainsi définir trois cas de figure, classés par difficulté croissante :

le contrôle est une variante d'un contrôle existant : on dit alors qu'on procède par héritage de ce contrôle. La classe définissant le contrôle devra alors hériter de la classe définissant le contrôle existant.

le contrôle est un "mix" de plusieurs contrôles existants : on parle alors d'assemblage. Cette fois, la classe de départ sera UserControl.

le contrôle est entièrement conçu à partir de rien, et la classe de base à utiliser sera Control.

Ces techniques dépassent assez largement l'ambition limitée de ce cours - et du temps qui nous est imparti. Ceux qui souhaiteront les mettre en oeuvre devront potasser des ouvrages spécialisés... et beaucoup plus onéreux que ce modeste site.

4. Quelques notions supplémentaires

Dans cette cette partie, je n'évoquerai les aspects techniques qu'au passage, voire pas du tout. Son but est autre : profiter de l'occasion pour introduire des notions supplémentaires de programmation objet, qui vous permettront d'aller plus loin dans cette démarche... si vous le souhaitez (ou si votre chef vous en donne l'ordre).

1. La surcharge

A l'instar de l'haleine, on entend parfois parler de **méthode surchargée**, ou de surcharge de méthodes. Kesako ?

Cela signifie que **la même méthode peut être appelée de diverses manières, c'est-à-dire avec des listes différentes d'arguments**. Cela se programme facilement : il suffit, dans la classe, de programmer plusieurs procédures du même nom, chacune ayant sa propre liste d'arguments. Chacune de ces procédures

devra comporter le mot-clé **Overloads**, signalant donc cette surcharge (c'est-à-dire signalant que si plusieurs procédures portent le même nom, ce n'est pas une erreur, mais au contraire que c'est fait exprès !)

Nous avons déjà rencontré, au moins à une reprise, une méthode surchargée : il s'agissait de **FromArgb**, qui, rappelez-vous, admettait au choix, trois ou quatre paramètres.

A signaler que lorsqu'une méthode d'une classe prédéfinie est surchargée, l'aide de VB.Net le signale systématiquement.

2. Membres statiques ou membres partagés

Ces **membres statiques** sont **des propriétés ou des méthodes d'une classe, qui existent et sont utilisables même quand aucun objet de la classe n'a été créé**.

Visual Basic offre en standard plusieurs classes qui disposent de propriétés ou de méthodes statiques, dont certaines que nous avons déjà rencontrées. Par exemple,

Les applications MDI

Pour finir ce cours en beauté, venons-en à la réalisation des applications poétiquement dites "MDI", à savoir : Multiple Document Interface. De quoi s'agit-il donc ? Tout simplement du lot commun des applications Windows bien connues, du genre Word, Excel, et tutti quanti. Toutes ces applications, malgré leurs différences, possèdent en commun une interface gérée par Windows, dans laquelle :

- il existe une fenêtre principale (celle de l'application proprement dite), appelée le plus souvent Form parente.
- cette fenêtre parente peut contenir un nombre quelconque de fenêtres appelées enfant. Chacune de ces Form enfant peut représenter un document différent, ou plusieurs vues différentes d'un même document.
- les menus existent exclusivement sur la Form parente, mais ils dépendent de la Form enfant active.
- les Form enfants peuvent être minimisées, agrandies, maximisées, disposées notamment en cascade ou en mosaïque

La réalisation d'une application MDI en VB.Net ne pose pas de difficultés exagérées, à condition, comme toujours, de procéder méthodiquement.

La première chose à faire sera bien sûr de définir la Form principale. Ceci s'effectue tout simplement en donnant la valeur **True** à sa propriété **IsMdiContainer**.

Passons à présent aux Form secondaires. Évidemment, dans la même application, et contrairement à la Form principale qui est par définition unique, il peut y avoir différents genres de Form secondaires, chaque genre étant adapté à un certain type de documents.

Mais même en faisant au plus simple, et en admettant que votre application n'en requiert qu'un seul genre, il subsiste tout de même un problème avec les Form secondaires : c'est que vous ne savez pas a priori combien il va y en avoir d'exemplaires à un moment donné de l'exécution. Si, par exemple, celle-ci est un traitement de texte simple, il y aura à l'intérieur de la Form principale autant de Form secondaires que de documents ouverts... c'est-à-dire de zéro à plusieurs dizaines.

Dès lors, la seule façon de procéder, pour les Form secondaires, sera de prévoir leur création dynamique, sur le modèle de ce nous avons déjà vu dans le chapitre 4 à propos de la création dynamique des contrôles. Il suffira donc de s'y prendre méthodiquement. Comme d'habitude, quoi.

Voici donc la marche à suivre :

1. Création de(s) Form(s) enfant(s)

On se contentera de créer une Form par type de Form enfant nécessaire. Cette étape ne nécessite aucune combine particulière : on crée la Form tout à fait normalement, en posant dessus les contrôles voulus, et en leur associant les procédures événementielles adéquates. Bref, jusque là, rien à signaler.

2. Instanciation des Form enfants

La chose à comprendre, c'est qu'au cours de l'application, les différentes Form enfants seront toutes des instanciations des Form enfants que vous venez de créer dans le paragraphe précédent. Dit d'une autre manière, jusque là, vous n'avez créé qu'un moule : il va maintenant falloir écrire le code nécessaire pour fabriquer les gâteaux.

Comme je l'ai déjà dit, ce code obéit aux règles déjà rencontrées lors de la création dynamique de contrôles. De plus, il faudra absolument signifier que la nouvelle Form ainsi créée est une Form enfant de la Form parent. Ceci s'effectue via la propriété **MdiParent** de la Form enfant (à noter qu'il s'agit d'une propriété accessible seulement par le code).

Ainsi, en admettant que nous ayons baptisé **Enfant** la Form servant de modèle à nos Form enfants, le code permettant de créer une nouvelle instance de cette Form sera :

```
Dim frm as New Enfant()
   frm.MdiParent = Me
   frm.Visible = True
```

Ceci suppose naturellement que ce code soit écrit dans la Form parent.

3. Maniement des Form enfants

Évidemment, la question ne se pose que lorsqu'on doit manier les Form enfant à partir de la Form parent. Le maniement des Form enfants à partir d'elles-mêmes ne pose aucune espèce de difficulté : programmer une Form à partir d'elle-même, c'est très précisément ce que nous avons fait depuis le début de ce cours.

La première chose à dire, et ce ne sera pas une surprise, c'est que les Form enfant font toutes partie d'une collection : la collection **MdiChild**. On pourra ainsi balayer l'ensemble des Form enfant par une boucle parcourant cette collection, ou rechercher une Form particulière au sein de cette collection en fonction de l'une ou l'autre de ses propriétés.

On notera pour terminer l'existence de la propriété de la Form parent **ActiveMdiChild**, qui renvoie la Form enfant actuellement active.